ECSE 2920: Design Methodology

Technical Documentation

Group #11

Spring 2025

Table Of Contents

Chapter 1 – Objective

Objective

The primary objective of this project is to design and develop an autonomous audio-tracking vehicle capable of identifying, locating, and driving toward specific audio frequencies. The system must also demonstrate controlled movement through predetermined paths, reliable signal detection and filtering, real-time motor actuation, and accurate response to environmental stimuli, all while being fully operable without user intervention during runtime.

To meet this challenge, the project integrates both hardware and software components into a unified embedded system. On the hardware side, the design includes: A Raspberry Pi 4 as the central control unit, a GPIO-controlled H-Bridge motor drivers for differential drive movement, a custom ±12V power regulation via push-pull amplifiers, filters for isolating audio tones (C6 and C8), a ADC for signal digitization, and Bump switches for collision detection.

On the software side, the vehicle is programmed using Python, leveraging libraries such as gpiozero for GPIO management, with modular code controlling Motor speed and direction, frequency detection algorithms, mode switching logic based on DIP switch input, Autonomous behavior execution on startup.

The project is structured around a series of incremental performance checkpoints, each adding to the overall capability of the car:

Checkpoint C – Figure Drawings:
The car must complete three distinct movement tasks including, travelling in a straight line for 3 feet and stop, driving in a square (3 ft x 3 ft) twice, and performing a figure-eight pattern twice, with 3-foot centerlines

Checkpoint B – Single Sound Detection:

The car must autonomously identify and move toward a C8 audio frequency (4186 Hz) while ignoring other distracting signals. The sound source is randomly placed within a 3–5 ft radius of the car, and up to two false frequencies may be present.
The car must successfully track the correct sound and stop upon triggering a bump switch, which indicates contact with the source. This must be completed twice in succession to pass the checkpoint.

Checkpoint A- – Dual Sound Detection:

Expanding on Checkpoint B, the car must now sequentially identify and track two audio frequencies, locating and driving to the C8 source, stopping upon bump switch activation and automatically begin searching for the C6 source (1046 Hz), and stopping again when contact is made. This dual-stage tracking must be reliably repeated twice to qualify for Checkpoint A-.

The end goal is to achieve full autonomous audio navigation, where the car can react in real-time to input conditions, perform algorithmic searches for peak signal strength, and reliably differentiate between multiple audio signals in a noisy environment.

Key Roles and Responsibilities:

The team members and their respective roles are as follows:

- **Michael See (Program lead):** Responsible for the overall programming required for this project. This member consistently pushes code to GitHub for other member's view.
- **Cade Toney & Finn Morton (Hardware Engineer):** Focuses on circuit design, GPIO configuration, and ensuring hardware stability. Leads tasks related to power supply and motor control.
- **Dawson Gulasa (Documentation):** Whilst also aiding the other members, this member consistently updates documentation. Whether that be the weekly deliverables, technical documentation, and user manual, this member takes the lead and ensures these documents are completed before submission.

Each member is responsible for documenting progress, participating in meetings, and contributing to design decisions to ensure project success.

Chapter 2 – Constraints

Constraints

The audio tracking car project was developed under a defined set of technical, physical, and evaluative constraints, which shaped decisions in both hardware and software. These constraints ensured the system aligned with the course's engineering learning objectives.

1. Component Usage

All active components—op-amps, transistors, comparators, H-Bridges—had to come from the provided course kit. While instructors permitted external components with approval, our design used only kit-supplied parts. This limited our options for signal conditioning and required creative use of basic analog components to achieve functionality without relying on specialized sensors or ICs.

2. Power Architecture

The project used two power domains:

- A 24V DC wall supply, from which we derived ±12V rails for the analog systems (filters and ADC).
- A separate USB adapter to power the Raspberry Pi 4, which also provided regulated 3.3V and 5V logic via onboard headers.

All analog rails had to be generated from the 24V supply, necessitating a custom push-pull amplifier, while isolating the Pi from voltage fluctuation or backfeed.

3. Audio Frequency Detection

To pass the core functional checkpoints, the car had to detect and respond to C8 at 4186 Hz ±500 Hz and C6 at 1046 Hz ±250 Hz

Detection had to occur autonomously, without user guidance. Multiple false frequencies were introduced during testing, so the car's bandpass filters needed to be highly selective and tuned to eliminate out-of-band noise. Audio signals were placed 3–5 feet from the car at random orientations.

4. Physical Space and Form Factor

The car's size limited circuit sprawl. Designs were graded partly on "look and feel," encouraging teams to build clean, compact layouts. Breadboards, wiring, and components needed to be arranged efficiently on the chassis to avoid clutter and ensure accessibility. This restricted component stacking, oversized power modules, and loosely routed signal paths.

5. Software & Programming Platform

All software had to be written in Python, executed on a Raspberry Pi 4. Signal interpretation had to rely on analog circuitry (filters) and basic GPIO-based ADC logic. Start-up behavior had to be fully autonomous: once a DIP switch configuration was selected and the car was powered, it had to complete the assigned task without further input.

6. Testing Environment Constraints

Testing was conducted under controlled, non-ideal conditions:

- Audio sources were placed at instructor-selected positions and angles.
- False signals were added to challenge frequency discrimination.
- Testing occurred in an auditorium, where sound reflections and echoes often caused spurious readings.

This forced us to carefully tune our filters and ADCs to reduce sensitivity to acoustic bounce and implement timing and direction correction logic in software to account for inconsistencies.

7. Manufacturing Readiness

Although a PCB was not physically built, all teams were required to create a fully manufacturable PCB schematic and layout, pass Design Rule Checks (DRC), generate a complete BOM and Gerber files, and submit to a vendor (JLCPCB) for quote confirmation. This placed real-world constraints on trace width, component selection, and layout clarity.

8. GPIO Resource Availability

The Raspberry Pi provided approximately 26 GPIOs, more than enough for our needs. However, the constraint encouraged efficient pin usage, especially for teams attempting more advanced ADC routing, DIP switch arrays, or dual-frequency signal paths.

Chapter 3A – Technical Design: Motors and H-Bridge

Purpose:

The purpose of the motors is to allow the car to move in the shapes laid out in the constraints (i.e. straight line, square, and figure 8) scan, and find the C8 and C6 signals. The motors are controlled via the H-Bridge and Raspberry PI and are powered by the buck converter (see appendix for more information on the buck converter).

The purpose of the H-Bridge is to allow the Raspberry PI and motors to interact with each other, allow for control of both speed and direction of the car.

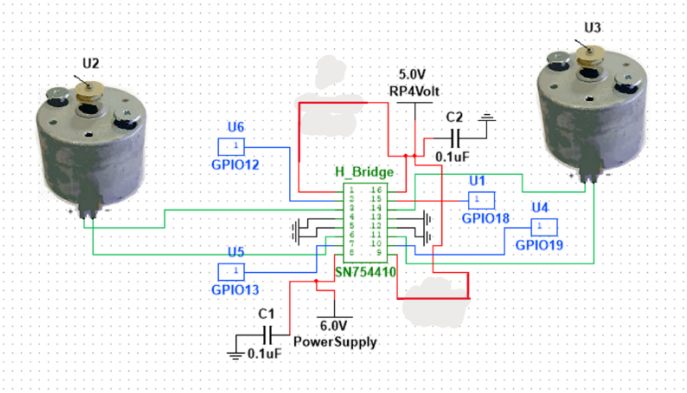
Research:

Research on the SN754410 IC, which is the IC containing the H-Bridge, was done via reading the data sheet and watching a YouTube video titled "DC Motor Control with an H-Bridge and Arduino (Lesson #17)" by the Science Buddies (see appendix A.A2 for link). This research led to a high-level understanding of the purpose of each pin of the IC, and how they should be configured in the circuit to achieve the project goals.

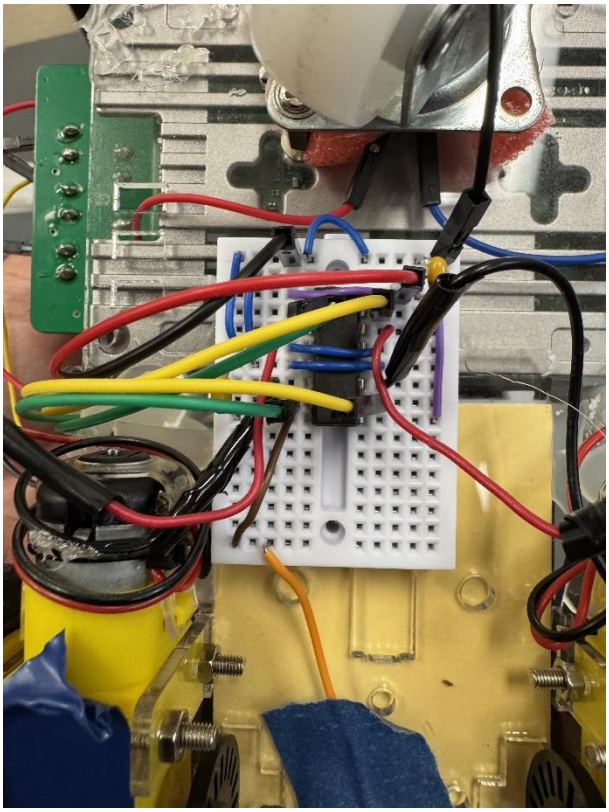
Previous Designs:

Immediately following the research of the design, the circuit was constructed and tested. The group only had 2 iterations of the H-Bridge circuit design, with one small difference between the 2. The first iteration had the enable pins connected to GPIO pins on the PI, while the second iteration had the enable pins connected straight to the 5V supply on the PI. See appendix A.A1 for the circuit diagram of the first iteration. This change will be explained in the testing and final circuit design of this chapter.

Final Circuit:



(**Figure 3.1:** Circuit for Final Iteration of H-Bridge.)



(**Figure 3.2:** Hardware for final H-Bridge iteration.)

This is the final iteration of the H-Bridge circuit for the audio car. Pin 1 on the H-Bridge serves as the enable pin for pins 2 and 7. This pin is connected to the 5V supply on the Raspberry Pi. This means that pins 2 and 7 are always enabled.

Pin 2 is connected to GPIO12 on the Raspberry Pi. When GPIO12 is high, the output at pin 3 of the H-Bridge—which is connected to the positive terminal of the motor—will be high as well. Pins 4, 5, 13, and 12 are all connected to ground and also function as a heat sink.

Pin 6, which is connected to the negative terminal of the motor, is controlled by pin 7 of the H-Bridge. Pin 7 is connected to GPIO13. When pin 7 is set to high, pin 6 will also output high.

Pin 8 is connected to 6V via the buck converter (see appendix A.A2) to supply power to the motor, while pin 16 is connected to a 5V source to supply logic power. Both pins 8 and 12 are equipped with 0.1µF bypass capacitors as recommended by the datasheet.

The remaining pins on the H-Bridge are configured in a similar manner to control an additional motor. The GPIO pins from the Raspberry Pi connected to the H-Bridge inputs operate the motor as follows: when one input pin is high and the other is low, the motor spins in one direction; reversing the high and low states causes the motor to spin in the opposite direction. If both pins are high or both are low, the motor will not spin.

Why This Design Was Chosen:

The only competing choices in the design was what to do with the enable pins 1 & 9. All other design choices are very standard and essential to the operation of the H-Bridge circuit. The group decided to wire the enable pins permanently to the 5V supply to save GPIO pins. The group knew they needed to save space for more intensive functions on the car later in the project and it would be best to conserve GPIO pins. The argument for keeping the enable pins controllable was to ensure stopping of the motors; however, this can be corrected with a python function that sets the control pins for one motor to both low.

The group also used GPIOs 12, 13, 18, & 19 because they have built in PWM functionality, which allows for smoother speed control of the motors. Both of these design choices will be further discussed in the testing section of this chapter.

Testing:

- Testing was done in a standalone environment using the benchtop DC power supply in the lab and the wave form generator.
- The circuit was constructed using the 5V and 6V from the DC power supply in the lab, and instead of using GPIO pins a 1kHz, 3.3Vpp square wave was used as the high input while ground was used as the low input.
- The group wired the enable pins to the 5V using the bench top supply.
- After the circuit worked as intended, the group made the design change of not using a GPIO for the enable and just putting it to 5V.
- The group also tested how the duty cycle of the square wave affected the RPM of the motor.
- The group started with a duty cycle of 50% and gradually increased it, and the motor speed increased as well.
- This is because the motors are experiencing higher average voltage with a higher duty cycle.
- The circuit was then tested using the buck converter supplying the 6V and the raspberry pi for the logic inputs and 5V supply.
- The circuit worked as intended and was complete for the purposes of this project.

Conclusion:

To conclude the group started the H-Bridge circuit design process by reading the datasheet and watching a YouTube video relating to the use of an H-Bridge for bi-directional motor control. The circuit was then built and tested in a standalone environment, which resulted in some minor design changes. After these design changes were made the circuit was completed and testing in unison with the buck converter and the pi and worked as intended.

Chapter 3B – Technical Design: Push Pull Amplifier

Purpose:

The purpose of the push pull amplifier is to change the 24V DC power supply from 0 – 24V to –12 – 12V. This is because the op amps used to power the filters need +/- voltage to properly amplify the signals from the mics. The push pull will do this by creating a virtual ground at 12V, and this will act as the new ground for the whole audio car. This will now turn 0V to –12V and 24V to 12V. The push pull amplifier will also need to be able to supply enough current to power the motors, because the group decided to only have one ground in the audio car circuit, so the ground loop for the motors will be through the push pull amplifier.

Goal:

Create a push pull amplifier that creates a stable virtual ground at 12V and supply enough current to run the motors.

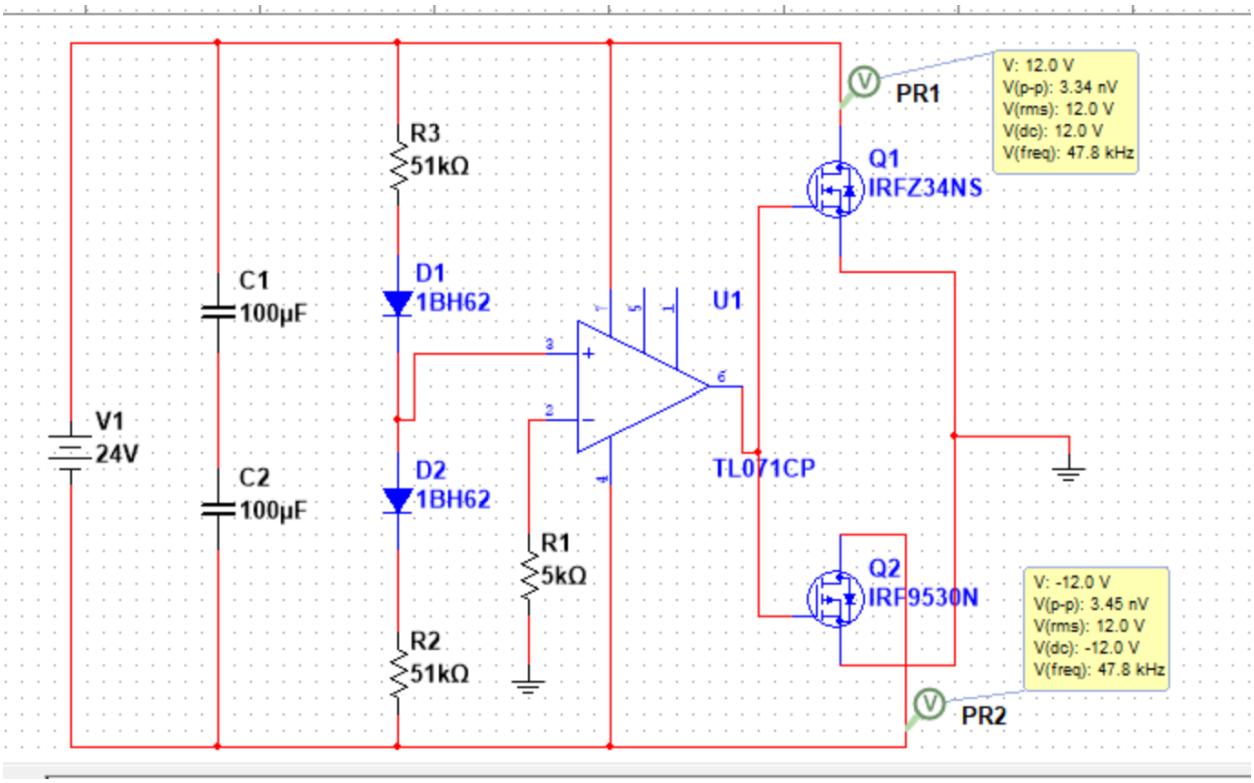
Research:

Research started with an article titled “Virtual Ground Circuits” (Appendix A.B2). This article contained a circuit titled Sijosae discrete rail splitter that was the inspiration for the first iteration of the group's push pull amplifier design. Once this design did not work the group found a reddit post (Appendix A.B2) that discussed using a push pull with an op amp in a buffer configuration to provide a stable virtual ground.

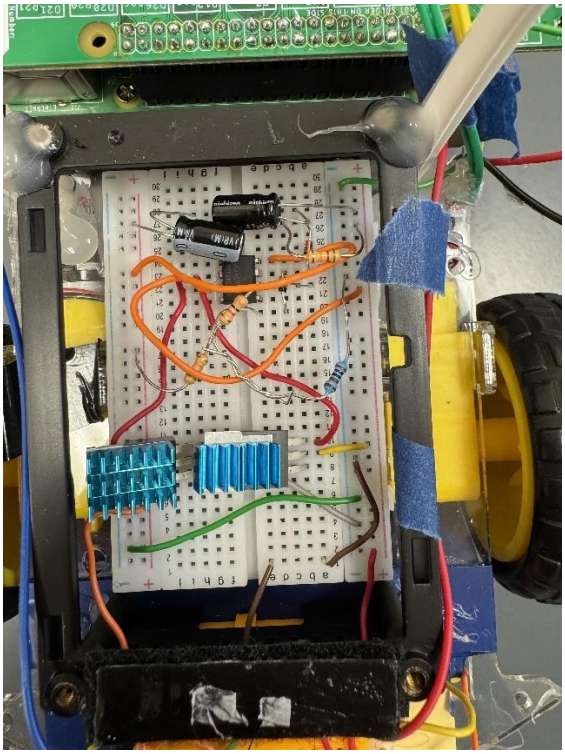
Previous Designs:

For the first push pull design iteration the group took the Sijosae discrete rail splitter as inspiration. The group replaced the BJT transistors in the design with MOSPHET and replace the 10Ohm resistors with potentiometers to allow for uneven loads. The circuit worked in simulation and in real life, to power the op amps. However, when the group shorted the PIs ground with the virtual ground supplied by this design it was not able to supply enough current to the motors. More information can be found in Appendix A.B1.

Final Design:



(Figure 3.3: Final push pull design schematic.)



(**Figure 3.4:** Final Push Pull design hardware.)

This is the final iteration of the group's push pull design. 2 main changes were made. The first was the position of the nmos and pmos were swapped, and the sources of the of the transistors both are wired to the virtual ground, and the drains of both transistors are wired to their respective voltage rail. The second big change was the inclusion of the op amp. The op amp serves as a buffer for the virtual ground keeping it a stable 12V regardless of the load. The mosfets can supply the proper current draw for the motors that the op amp cannot.

Why This Design Was Chosen:

Ultimately the group chose this design because it is capable of creating a stable virtual ground and supplying the current needed to run the motors.

Why This Design Works:

This circuit is best understood by splitting the circuit into 2 parts. The op amp and the transistors. The op amp provides stable 12V for the virtual ground, and the mosfets help supply current to the load.

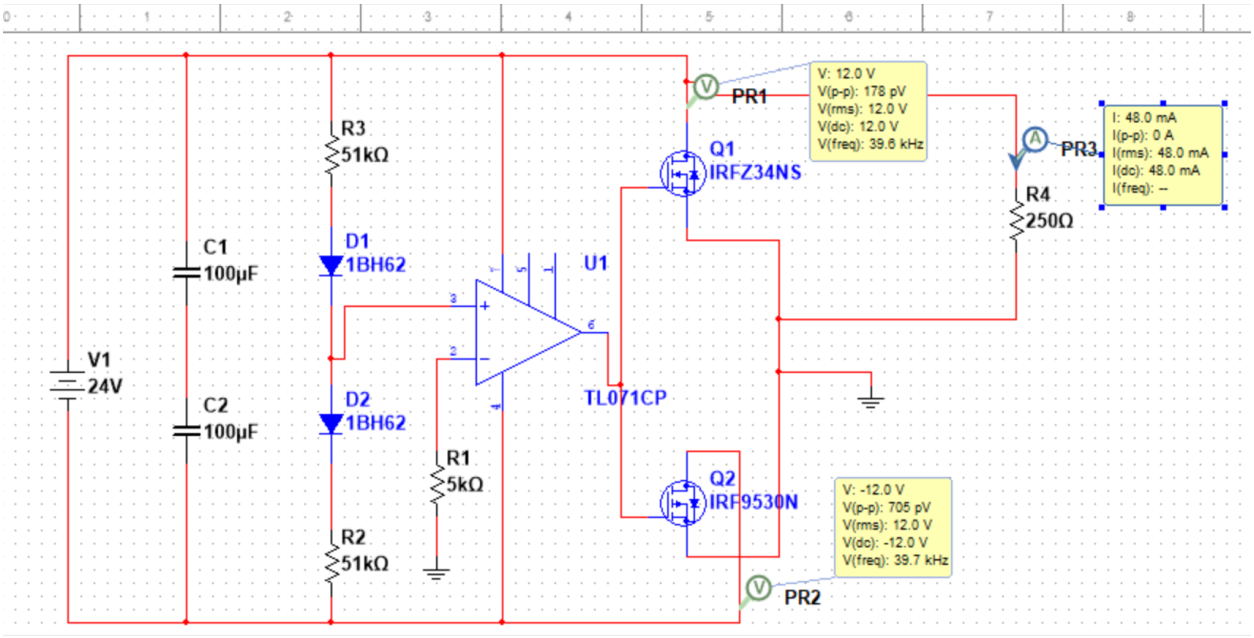
This design works by using a resistive voltage divider to set the voltage to the non-inverting input of the op amp. The 51kOhm resistors are in series because there is no current flow into the op amp, and regardless of which diode model one uses for the analysis the node between the 2 diodes will always be half the voltage of the DC supply (12V for the 24V supply), as long as the resistors and diodes are identical (and assuming the diodes are on). The op amp has negative feedback, even though there are transistors in the feedback loop. This means that the non-inverting and inverting input have the same voltage. No current flows into the op amp so the voltage drop across the 5kOhm is 0. The inverting input is shorted to the virtual ground; this completes the feedback loop and sets the virtual ground at 12V.

For both the n-channel and the p-channel mosphet the gates are both connected to the output of the op amp and the sources are both connected to the virtual ground. This is important because both mosphets have the same Vgs, which controls whether or not current can flow from the drain to the source. For current to flow from the drain to the source, Vgs > threshold voltage for an n-channel, and –Vgs < threshold voltoge for a p-channel. Assuming the mosphets have the same threshold voltage when one is on the other will be off. How does this help for higher current draw?

For example, if one connects a 5k Ohm load between the positive rail and virtual ground, and one makes the assumption that the n-channel is off. All of the current will flow into the 5KOhm. Then because the n-channel is off the p-channel must be on the current will flow through the virtual ground into the drain of the p-channel, and because the p-channel is on current will flow from the drain to the source which is connected to the negative side of the battery completing the current path. See appendix A.B3 for a more in depth paper analysis.

Testing:

- When testing the first iteration of the push pull, it was tested in a standalone environment using the DC bench top supply in the lab and using a simple resistive load.
- It was also tested using the 24V 5A supply from the wall and an op amp load.
- Both worked; however, when the ground from the Pi and virtual ground were shorted, the push pull could not supply the current needed to drive the motor.
- The group then researched the second push pull design and tested it in simulation and then with the raspberry pi and the motors and it worked.
- Simulation results from the second design are shown below:



(**Figure 3.5:** Simulation results for final iteration of push pull.)

- This shows the stable virtual ground and large current output

Conclusion:

To conclude the group started with researching buffered virtual ground circuits and this led to the first iteration of the push pull amplifier. It worked up until connecting it the car where it failed to drive the motors. Through trouble shooting, more research, and testing in simulation the group was able to land on a design with an op amp. The group tested the circuit in simulation and physically in conjunction with the raspberry pi, filters, and motors. The design worked and became the group's final iteration of the push pull amplifier.

Chapter 3C - Technical Design: Filters

Purpose

The filters in our project serve the essential function of isolating specific audio frequencies, namely, C8 (4186 Hz) and C6 (1046 Hz), from surrounding environmental noise. This isolation ensures that only target frequencies are passed to the ADC for conversion, enabling the car to identify, localize, and respond to audio signals in real time. For Checkpoint B, the car needed to detect a single sound frequency (C8). To qualify for Checkpoint A, it must distinguish between both C8 and C6, track them in order, and ignore all other sounds and frequencies.

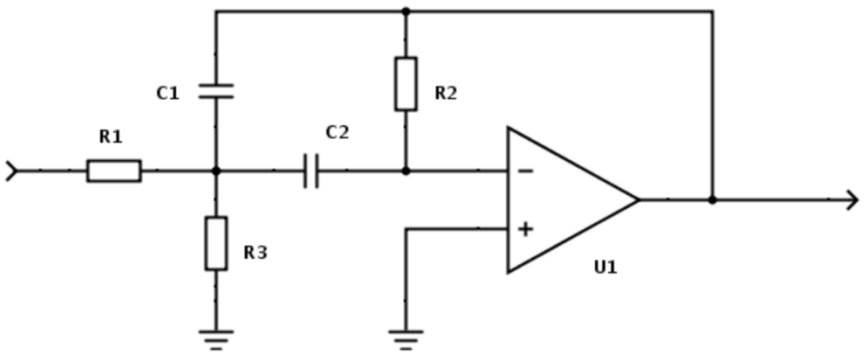
Research

Initial microphone characterization and frequency response testing (using both an oscilloscope and spectrum analyzer) made clear that the raw signal needed significant refinement to identify a narrow frequency reliably. Passive filters were ruled out early due to inadequate gain and insufficient selectivity. From here, the team explored active bandpass filters, using online design tools to match calculated component values to our required center frequencies.

The final design was heavily informed by a resource used to calculate and simulate narrow bandpass filters (Reference Appendix chapter **A.2**)

Key considerations extracted from this research:

- Using TL084 quad op-amps to implement active second-order filters
- A single stage of a narrow bandpass filter blocks low and high frequency noise
- A rectifier + smoothing capacitor stage is needed to prepare signal for ADC use
- Matching gain to our ADC's voltage reference range is critical to detect differences between 0–3 logic levels

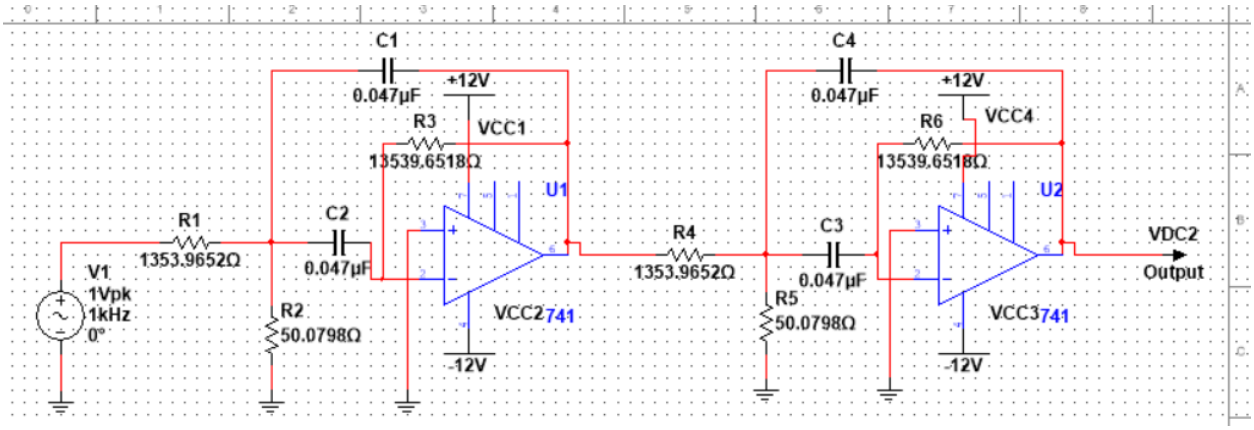


(**Figure 3.6:** Circuit Diagram for chosen Narrow Band-Pass Filter used to Target both C6 and C8.)

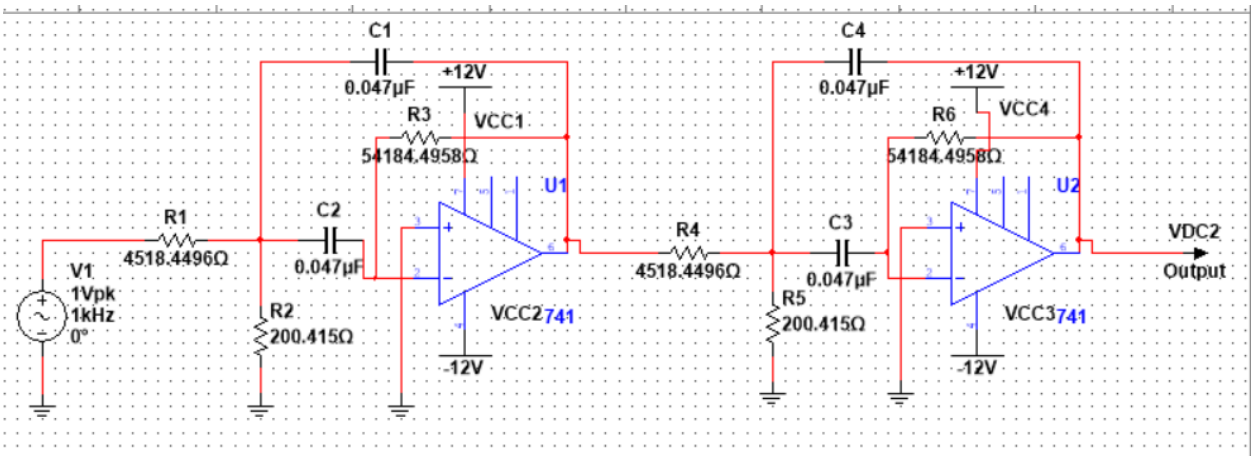
Final Design

To ensure signal strength and frequency precision, we implemented two narrow bandpass filter stages in series for each frequency channel (C8 and C6). Both stages use the same resistor and capacitor values, effectively reinforcing the frequency window while suppressing out-of-band noise.

- C8 Filter Center Frequency: 4186 Hz
- C6 Filter Center Frequency: 1046 Hz
- Gain: ~5
- Q-Factor: ~8.3



(**Figure 3.7:** Final Filter Topology Targeting C8.)



(**Figure 3.8:** Final Filter Topology Targeting C6.)

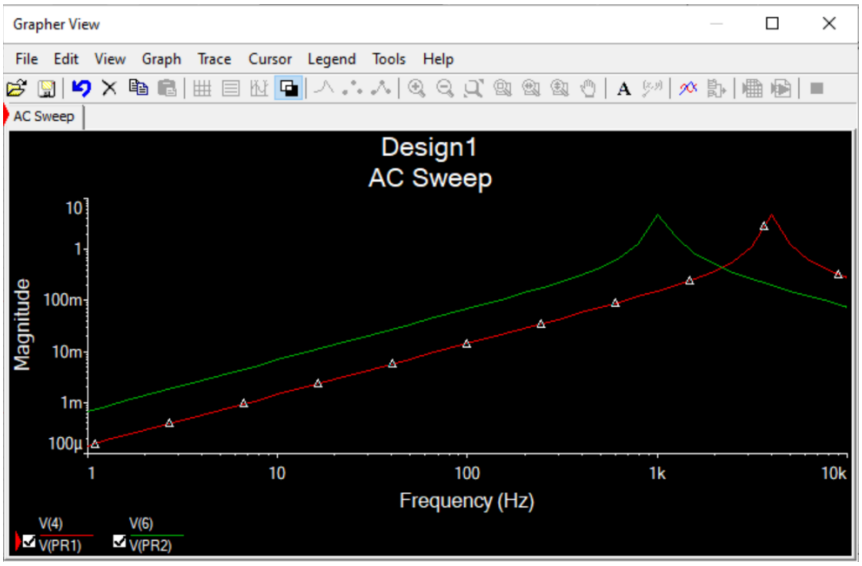
The mathematical calculations for the resistor and capacitor values can be found in the appendix. (Reference Appendix Chapter **A.3**)

Simulations

We used Multisim to simulate the frequency response for both filters:

- Each filter alone showed a strong peak at its intended center frequency.
- Cascading two identical filters in series sharpened the frequency band.

The final simulated Bode plot confirmed that the C8 filter has maximum gain at 4186 Hz, with sharp roll-off and that the C6 filter peaks at 1046 Hz, with similar performance.



(**Figure 3.9:** Combined Bode Plot for Filters Targeting C6 and C8 Frequencies.)

Why This Design Was Chosen

We selected the active narrow bandpass filter topology based on its:

- Proven success in early single-frequency testing (Checkpoint B)
- High gain and narrow bandwidth (critical for frequency discrimination)
- Flexibility to cascade multiple identical stages for greater selectivity
- Stable behavior under varying input volumes

Our choice to use two stages per frequency was driven by physical test data. While one stage isolated the tone sufficiently for early demos, only with two cascaded stages could we eliminate false positives introduced by background noise and overlapping harmonics.

Why This Design Works

By precisely tuning R and C values using the bandpass filter formula:

$$fr = \frac{1}{2\pi\sqrt{R}1R2C1C2}$$

We achieved highly accurate frequency targeting.

The op-amps in each filter stage boost the signal and maintain stability, the rectifier and capcitor (placed after each dual-filter chain) then ensures that the analog waveform is smooth and positive-only before ADC input. The cascaded structure does not attenuate the signal excessively due to the built-in gain.

In effect, our design reliably extracts only the signal components around C6 and C8, filters out background noise, and delivers steady analog voltages suitable for comparator-based ADC conversion.

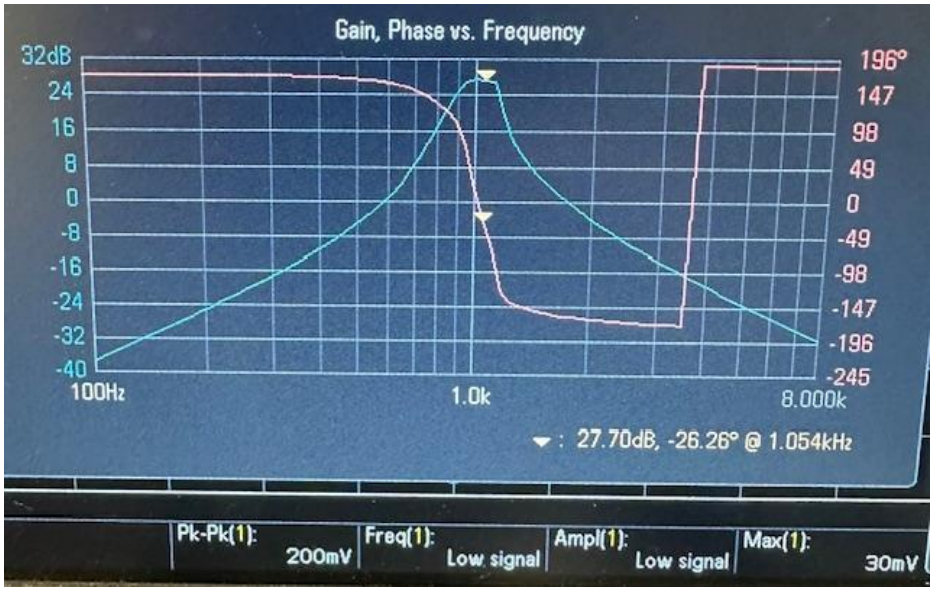
Physical Testing

Once simulated and built, we tested each filter stage with:

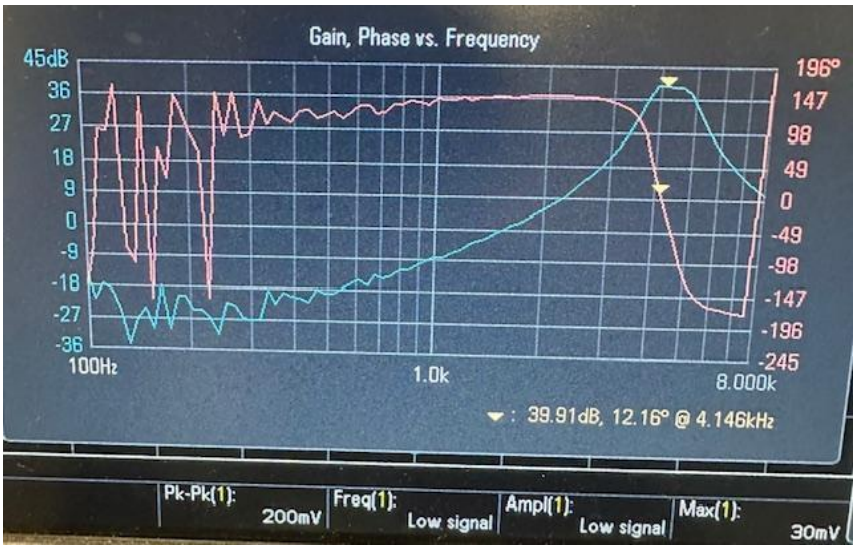
- A function generator, sweeping from 500 Hz to 5000 Hz
- A microphone input, with real audio tones played through a speaker
- A spectrum analyzer, verifying voltage output vs. frequency input

We validated the following:

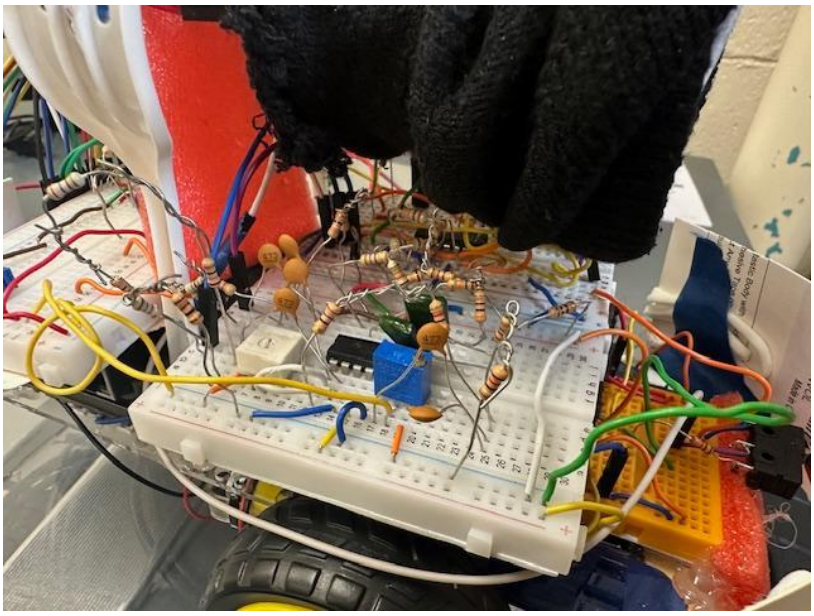
- Each C8 filter pair only amplified tones ~4186 Hz ±250 Hz
- Each C6 filter pair only amplified tones ~1046 Hz ±250 Hz
- Output voltages post-rectification reached usable ADC voltage thresholds
- When connected to the full ADC chain, signal strength outputs matched expected behavior for signal detection



(**Figure 3.10:** Bode plot for C6 With Function Generator Input.)



(**Figure 3.11:** Bode plot for C8 With Function Generator Input.)



(**Figure 3.12:** Final Filter implementation on Audio Car Chassis.)

Conclusion

The dual-stage active narrow bandpass filters for C6 and C8 are a core part of our car's audio tracking success. Their design was validated in simulation and physical tests. With strong gain, precise frequency targeting, and reliable integration with our ADC system, these filters give our project the fidelity required to detect and differentiate sound sources, making them essential for reaching Checkpoint A and building a functional dual-frequency tracking car.

Chapter 3D – Technical Design: ADC

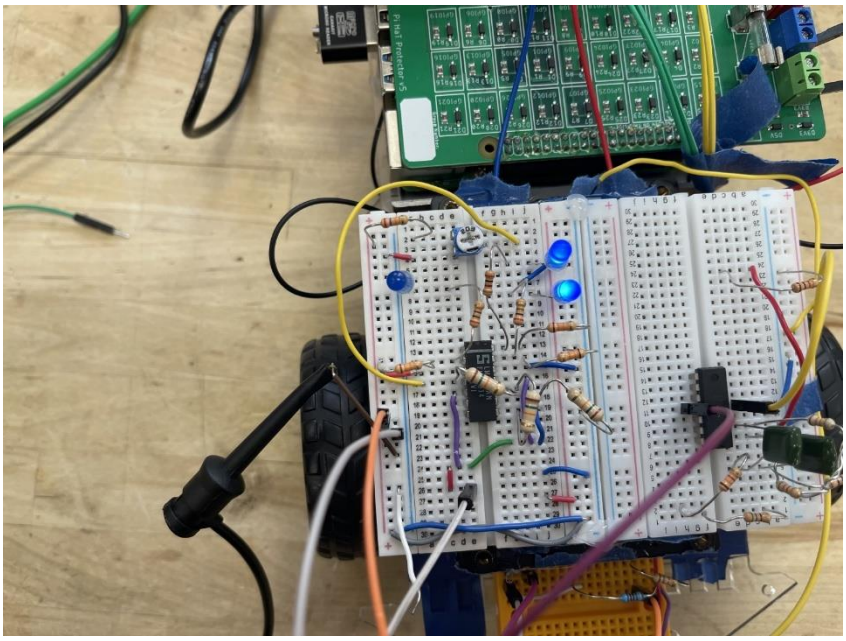
Purpose

The purpose of the ADC is to convert the C8 and C6 signals into digital numbers which can be implemented into the code of the car. This allows the car to see the difference between the target and fake frequencies.

Research

Since we are using the LM339, it requires a pull-off resistor in the output. The initial R-2R ADC design was dysfunctional because of this information. It is important to make sure that the GPIO isn't overvolted and only gives around 3.3V. This was achieved through the resistor ladder design with 4 resistors. A potentiometer would also create an adjustable voltage divider among the reference voltages which could readjust the sensitivity of the car.

Schematic and physical picture



(**Figure 3.13:** Physical ADC placed on Audio Car.)

Why This Design Was Chosen

We chose this design because the 4 resistors in the resistor ladder each have a node compared to the reference voltage. Then each comparator output, without logic, represents a decimal count of the strength. If 2 comparators are on, then we have a strength of 2. If 1 is on, when we have a strength of 1. While this number format isn't binary, since the resolution is so small, we only need an extra GPIO to count from 0 to 3. This will allow the car to record maxes (spikes) in the frequency readings and stay consistent in different environments. The potentiometer can also be adjusted to change the sensitivity of the car to compensate for said environments.

Why This Design Works

Since the Pi will only be receiving digital signals, time will be the best reference when it comes to giving context to those signals within the software side. The car will do a full 360 to record the highest value. Once that value is found it will now turn in the same direction and record the amount of time that the strongest strength is being read for. That algorithm is for the C6 frequency. For the C8, we used a different algorithm where the car will do a full 360 to record the highest values. It will then detect how much time it took to find the frequency and turn back towards the target depending on how long it took to find the target initially. These algorithms allowed the car to get precise and accurate readings.

Testing

What was done:

- Physically build the ADC using the +/- 12V supply
- Apply a known analog voltage to the ADC and measure the expected output
- Demonstrate ADC functionality using a multimeter and known input values
- Provide the ADC topology and calculations showing input-output mapping
- Explain the resolution of the ADC and how it will be used for turning the audio car

The ADC was powered with the push/pull and used a DC input as the input signal. It was then mapped at three sensitivity levels and was matched to the output voltage bode plot from our filter.

We measured the output voltage of the ADC in respect to the raspberry pi ground.

Conclusion

A potentiometer-based adjustable voltage divider was chosen for the ADC reference voltage, allowing on-the-fly calibration in different environments. The ADC was tested with a push/pull power setup and a DC input, confirming that it produces expected digital outputs. The ADC successfully maps signal strength which allows for good readings on the microphones and filters. This worked well with our algorithm and other hardware designs which allowed the ADC to be a success because of our adjustable voltage divider and precise filters.

Chapter 3E – Technical Design: Hardware Collaboration

Introduction:

This sub-chapter will cover hardware collaboration. It will look different from most sub-chapters within this chapter, because it will cover less design choices and design iterations and more how every piece of hardware connects and interacts with each other to achieve the project goals.

This sub-chapter will cover the path the audio signal is taking to reach the PI in a way that code can interpret it and be able to locate and drive towards the signal source. As well as how the motors and H-Bridge are connected to the whole circuit.

Signal Path:

Sine wave signals are played via a phone app. The 2 microphones on the car, one for C8 and one for C6, pick up these signals and convert the mechanical energy of the sine wave into an electrical sine wave. The mics have a 5V pin that is powered by the PI, a ground pin, and a signal pin. The ground pin is connected to the one ground in the circuit that is the virtual ground provided by the push pull amplifier. This ground is the same as the PI ground. The signal pin from one microphone is wired to the input of the C8 filter, and the other to the C6 filter. The filters are active filters and have op amps to provide gain. The Vcc+ and Vcc- pins of the op amps are powered by the 12V and –12V from the push pull amplifier. This allows the op amp to amplify the signals in the positive and negative direction capping out a magnitude of 12V. The output of the C8 and C6 filters are input into 2 different half bridge rectifiers that are a diode and a resistor in series. The resistor goes to ground and the voltage across the resistor is input into 2 2-bit ADCs respectively. Each 2bit ADC has 4 quantization levels and a GPIO pin is connected to the output of the ADC at each different quantization level. Using software, the car can then interpret the output of the ADC and locate the audio source.

Motors and H-Bridge:

As discussed in an earlier sub chapter, the motors require 6V in order to operate. This 6V is provided by the buck converter. The input of the buck converter is the 12V from the push pull amplifier that is then stepped down to 6V via the buck converter. The ground for the H-Bridge is the virtual ground provided by the push pull amplifier. The H-bridge logic is controlled by GPIO pins from the PI, where based off the algorithm for finding the signal the PI will make the motors spin and drive towards the source.

Testing:

- While each individual component was tested in isolation, the results of which are covered in their respective sub-chapters.

- Testing was also conducted in accordance with all the hardware connected in collaboration with software.
- One of the main tests ran was ensuring proper code interpretation of the signal.
- The group would play the sound signals into the microphone of the car while a debug program would output the interpretation of the ADC to the console.
- Most of the time these values made sense and confirmed our hardware was working well altogether.
- However, there were inconsistencies which led to the doubling up of the group's filters, adding a second ADC, and moving away from a transistor switch.

Conclusion:

Overall, this chapter covered how the hardware in the car collaborates which each other to locate and drive towards the sound source. This was done by discussing the signal path how it starts with the microphones, then filters, then rectifier, then ADC, and lastly the raspberry PI. How the motors connect was also discussed; how the same push pull that powers filters, powers the buck converter, that powers the motors that are controlled by the PI via the H-bridge.

The group also discussed how testing the hardware collaboration was conducted, what the results were, and how this led to design changes.

Chapter 3F – Technical Design: PCB

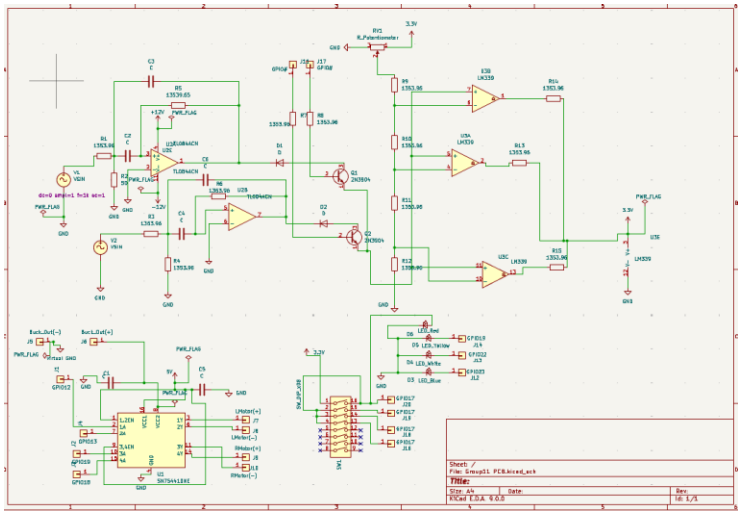
Purpose

The purpose of the PCB design is to simplify the initial design by minimizing the wires and spaces the circuits take up by fitting everything on one board. There are also no loose wires in a PCB which prevents wires from coming out and potentially any short circuits from happening. The layout prioritizes short and direct traces from the filters to the ADC to reduce noise susceptibility. I/O ports are placed on the board edge to simplify wiring to external components (e.g., microphones, motors, and RP4 GPIO headers).

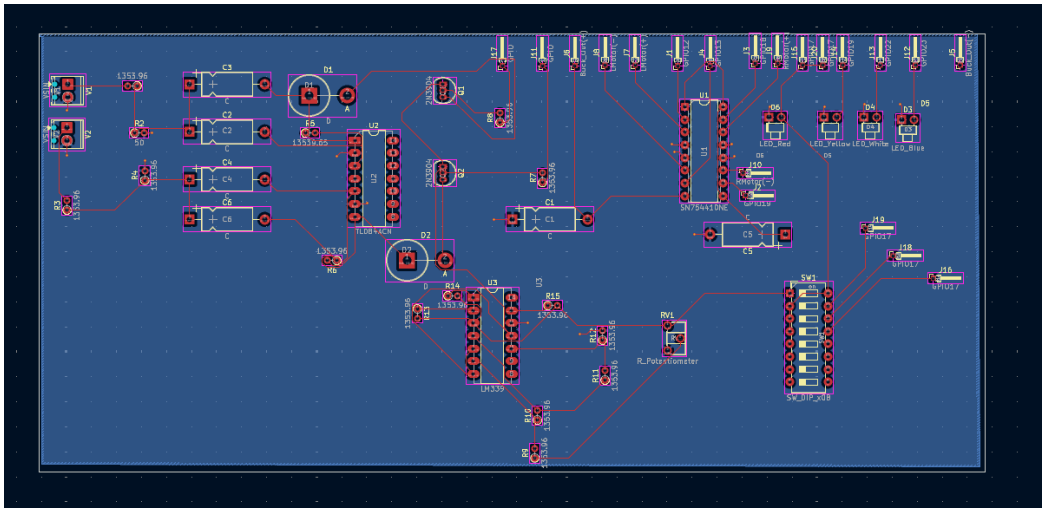
Research

The group did not have a designated member with strong PCB experience. So, we decided to use KiCad which is an easy-to-use free PCB software. It took lots of time figuring out the program and many problems were faced like having multiple filters reference the same component in a circuit, so it translates to the 3D design properly. All power in the inital schematic must be connected to a power flag for the program to recognize the component as a power source. In the 2D design, traces must be clean and precise without overlapping each other. There must also be a bottom layer that references ground that all ground nodes can be referenced to. You must create this plane yourself by selecting the zone and filling it. Make sure all of this passes the DRC once you are done.

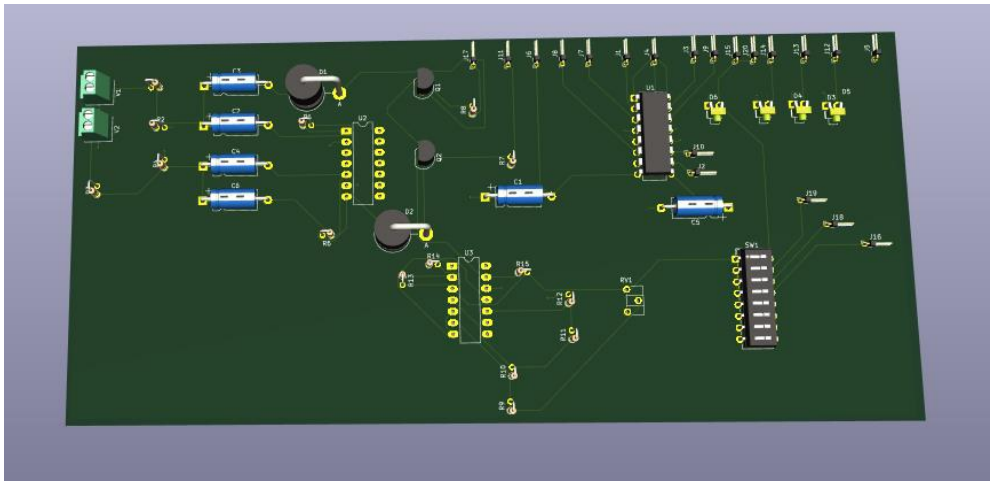
Schematic



(**Figure 3.14:** PCB Schematic in KiCad.)



(**Figure 3.15:** PCB Board Layout.)



(**Figure 3.16:** 3D Render of Board.)

Why This Design Was Chosen

This design combines the schematics of all finalized designs in our project. If we had printed out our PCB, we could then solder it and not have to worry about loose parts within our breadboards. The space the breadboards take up would also be minimized. Short and direct traces reduce noise susceptibility which allows for more accurate readings.

Why This Design Works

All points are connected to a ground layer which allows for less wiring while all components have the same ground reference. The DRC checker makes sure that we have a working circuit. The PCB design references the already completed and working design using bread boards which means theoretically the PCB will work.

Testing

- Ensure all components used are purchasable and compatible
- Pass all Design Rule Checks (DRC) within the software
- Make sure all designs used in PCB are the finalized designs

Conclusion

The PCB schematic and layout meet all project requirements. Traces are not very large because of lower current. Each subsystem was modeled to reflect the existing breadboarded circuit design and then optimized for real-world fabrication. All components used are standard through-hole or DIP, ensuring simple soldering and debugging. This PCB will serve as the base for signal analysis and control logic in our audio-seeking vehicle.

Chapter 4A – Software: Structural Overview

Purpose

The purpose of maintaining a form of structure in our software is primarily to organize our work in a way that is understandable and reasonable to work with on a daily basis for anyone in our group. This applies to both the tree of directories that contain our python programs and the structure of the programs themselves. If an additional python file is to be implemented, it must serve a clear, definitive purpose that's unique from what's already in place. This approach applies to the functions inside each file as well.

Possible Approaches

When tackling structure and organization, we face a similar conflict to "Top Down vs. Bottom Up", but for the context of this scenario, we can label it as "Over-organization vs. Under-organization".

Hypothetically, the Audio Car could function identically if we had every relevant action inside a single python file. This file would use no functions, and everything would be written and implemented in the moment. The result of this would be a 1000+ line file that contains multiple repeating segments and an inadequate environment for anyone who needs to debug/change anything inside this file. This is under-organization.

We could also have a fleet of python files, each containing a single function. These could be spread through a sprawling directory tree as well. We would have 20+ files, and each file would contain no more than 50 lines of code. Unfortunately, a large portion of that code will be importing other files. While each file is relatively easy to modify and debug, it is now the journey to get to each file that is tedious and inefficient for users to access. This is over-organization.

Like the relationship that "Top Down" and "Bottom Up" share, over and under-organization come with their respective shining pros and glaring cons, and generally, it is best to take bits and pieces from both approaches and meet somewhere in the middle.

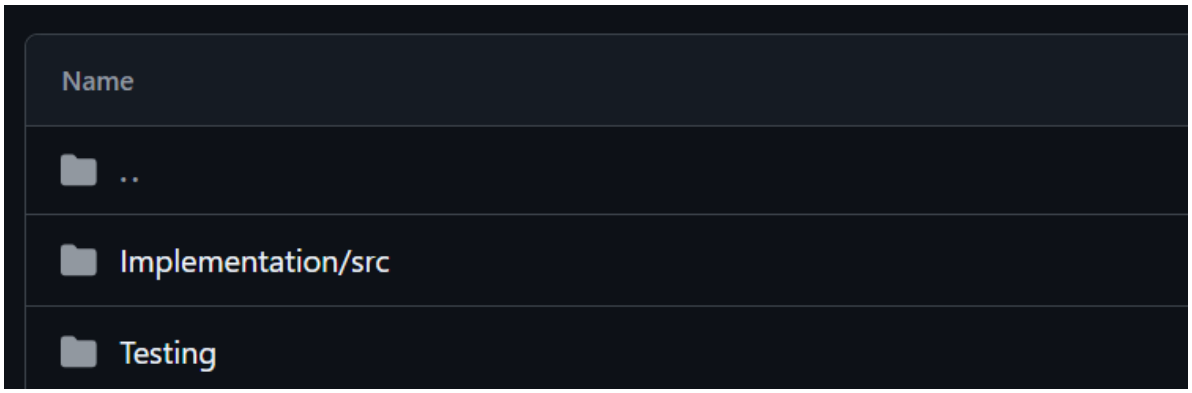
Chosen Approach

We have chosen to find a balance to our code organization. Because this is a small-scale project on the software side, we don't need too many individual files, but we definitely still need a degree of separation between the different actions that the car is capable of. We have 2 directories named *Testing* and *Implementation*. Any files being used in the Audio Car's intended operation will be placed in Implementation while any files used to test/calibrate components during development will be placed in Testing.

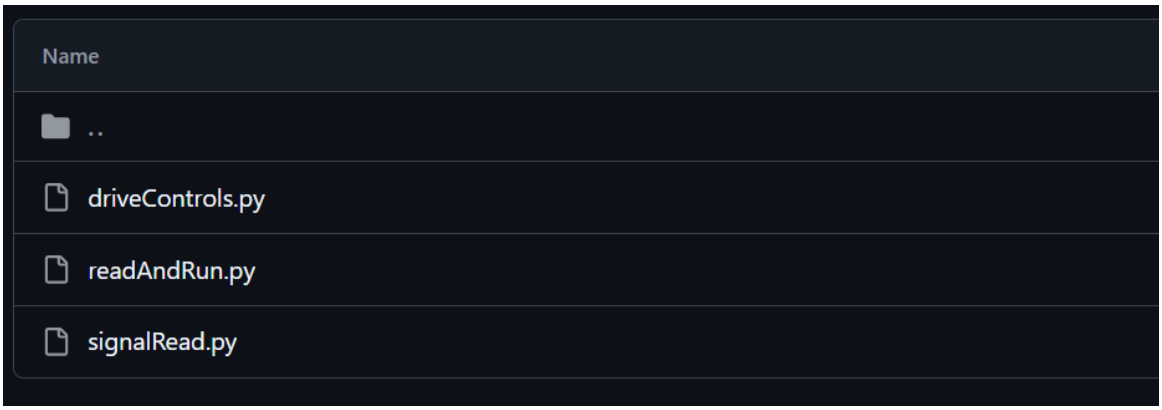
Inside the Implementation directory, we have 3 python files that serve a distinct purpose. The first file, driveControls.py, is used to handle motor movements. It includes primitive functions such as turning left 90 degrees and moving straight for x number of seconds, where x is a function parameter. These primitive functions build upon each other to create more complex movements such as driving straight for 3 ft, making the shape of a square, and moving in a figure eight pattern.

The second file, signalRead.py, is used for the “Single” and “Multiple” modes where the car’s behavior relies on the strength of a target audio signal. It includes simpler functions like *clock*, which gathers the sum of ADC values and averages those readings over a specified number of iterations. These simpler functions are then used to build more complicated functions like “checkpointB” and “checkpointA”, which are our algorithms for finding 4186Hz and 1046Hz respectively. Both functions use the clock function regularly to approximate the location of their target audio sources. Most smaller functions are only called inside of the file they’re defined in, while the larger, more complicated functions are usually called from inside the third python file. This file also occasionally uses commands from driveControls.py in its main “Single” and “Multiple” functions.

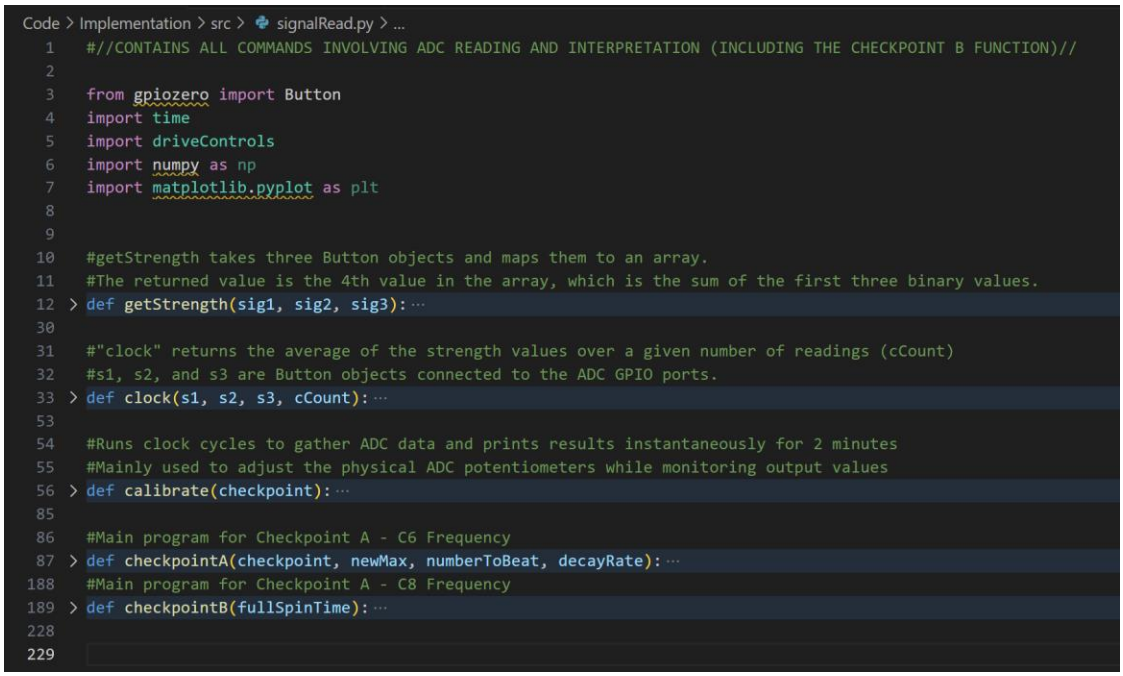
The third file, readAndRun.py, can be seen as the “main” program. It acts as the bridge between the user and the car’s available actions. It consistently monitors the values of the rear dipswitch and calls their respective commands when conditions are met. These commands are generally the most complicated, high-level functions residing in the other two files, which keep the main program clean, readable, and relatively easy to debug. We also make sure to implement inline commenting and comments for each function in order to maintain understandability.



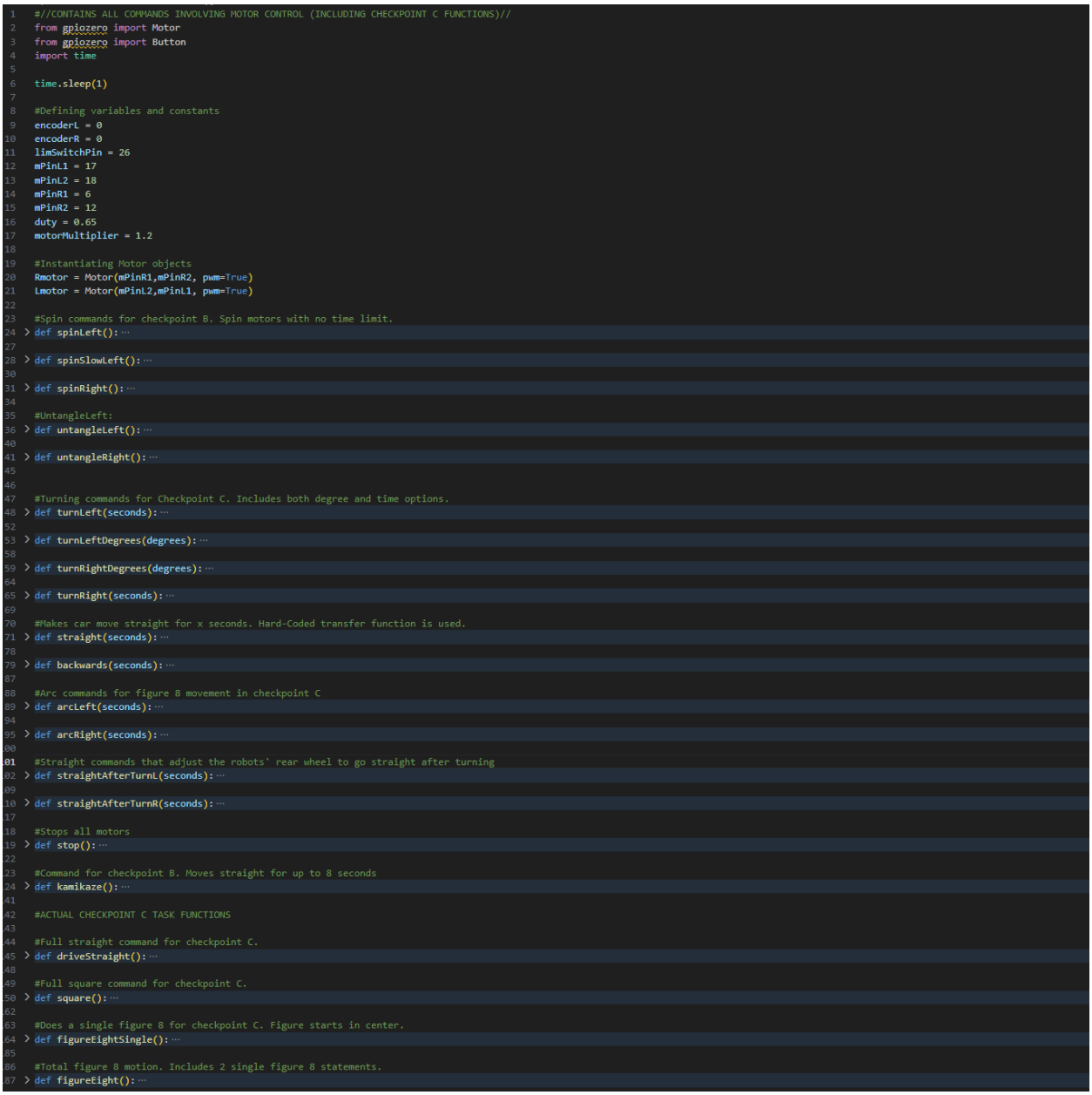
(**Figure 4.1:** Display of Code Directory on GitHub)



(**Figure 4.2:** Display of Code/Implementation Directory on GitHub)

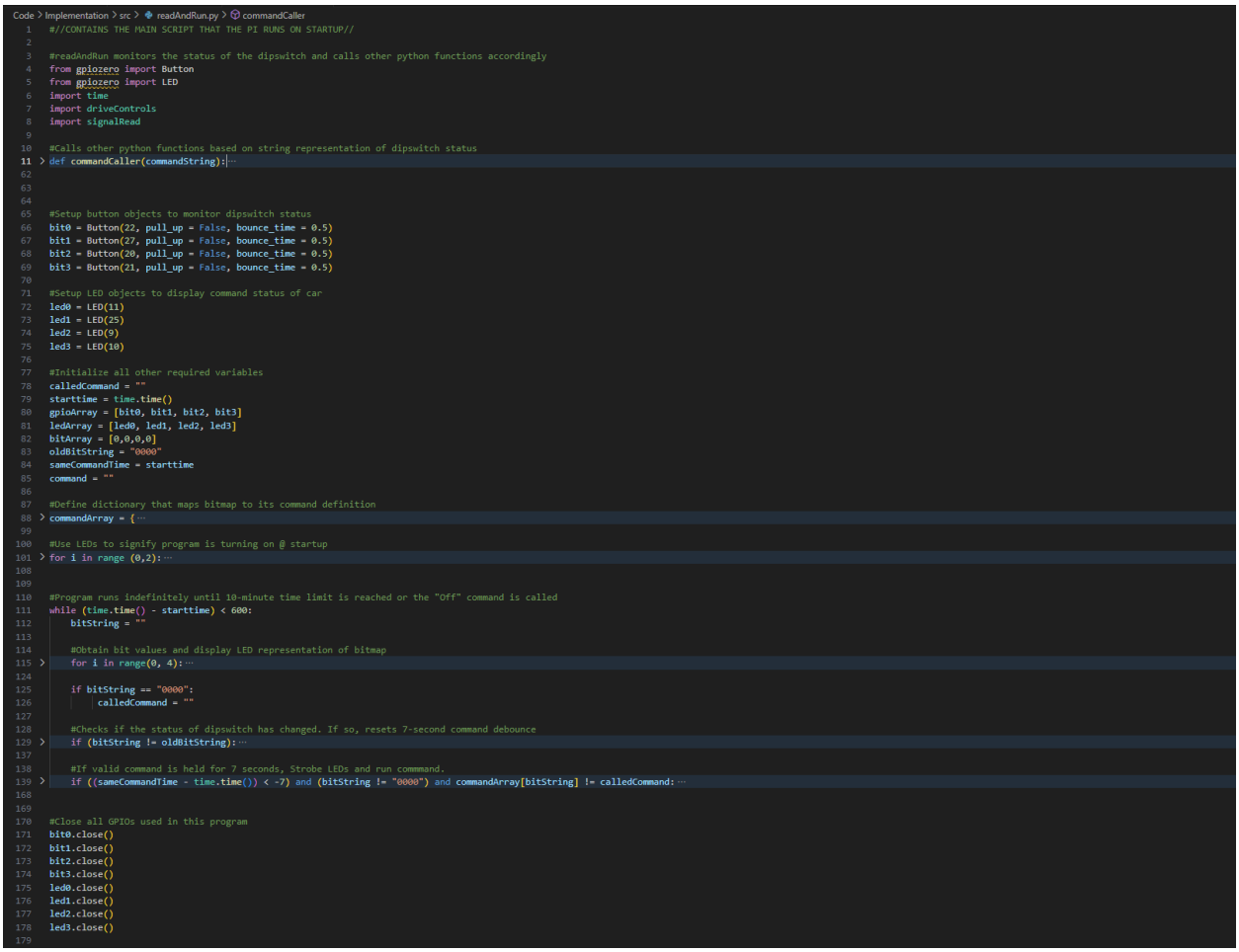


(**Figure 4.3:** Collapsed View of signalRead.py)



```
#//CONTAINS ALL COMMANDS INVOLVING MOTOR CONTROL (INCLUDING CHECKPOINT C FUNCTIONS)//
from gpiozero import Motor
from gpiozero import Button
import time

time.sleep(1)

#Defining variables and constants
encoderL = 0
encoderR = 0
limSwitchPin = 26
mPinL1 = 17
mPinL2 = 18
mPinR1 = 6
mPinR2 = 12
duty = 0.65
motorMultiplier = 1.2

#Instantiating Motor objects
Rmotor = Motor(mPinR1,mPinR2, pwm=True)
Lmotor = Motor(mPinL2,mPinL1, pwm=True)

#Spin commands for checkpoint B. Spin motors with no time limit.
def spinLeft(): ···

def spinSlowLeft(): ···

def spinRight(): ···

#UntangleLeft:
def untangleLeft(): ···

def untangleRight(): ···


#Turning commands for Checkpoint C. Includes both degree and time options.
def turnLeft(seconds): ···

def turnLeftDegrees(degrees): ···

def turnRightDegrees(degrees): ···

def turnRight(seconds): ···

#Makes car move straight for x seconds. Hard-Coded transfer function is used.
def straight(seconds): ···

def backwards(seconds): ···

#Arc commands for figure 8 movement in checkpoint C
def arcLeft(seconds): ···

def arcRight(seconds): ···

#Straight commands that adjust the robots' rear wheel to go straight after turning
def straightAfterTurnL(seconds): ···

def straightAfterTurnR(seconds): ···

#Stops all motors
def stop(): ···

#Command for checkpoint B. Moves straight for up to 8 seconds
def kamikaze(): ···

#ACTUAL CHECKPOINT C TASK FUNCTIONS

#Full straight command for checkpoint C.
def driveStraight(): ···

#Full square command for checkpoint C.
def square(): ···

#Does a single figure 8 for checkpoint C. Figure starts in center.
def figureEightSingle(): ···

#Total figure 8 motion. Includes 2 single figure 8 statements.
def figureEight(): ···
```

(**Figure 4.3:** Collapsed View of driveControls.py)



(**Figure 4.4:** Collapsed View of signalRead.py)

Testing

- Ensure that the Code directory is relatively easy to navigate for all group members
- Ensure that each program is easy to navigate and decipher
- Maintain a sense of purpose in organizing/separating software files

Conclusion

Because we decided on our organization strategy early on, everyone on the team was easily able to look through our code and understand the functionality and purpose of each line. As a result, hotfixes could be done in seconds, advice could quickly be supplied, and new ideas could be prototyped with ease. We believe that there was an efficient, yet robust amount of organization applied to our software and that it was highly beneficial to the development and performance of the Audio Car.

Chapter 4B – Software: Functional Overview

4B.1 - Motor Controls

Purpose

The Audio Car will require motor movement for every task available to the user. This includes actions such as:

- Making a square
- Drawing a figure eight
- Spinning to scan for an audio source

These “higher-level” actions will require smaller tasks such as:

- Driving in a straight line
- Turning with one wheel
- Stopping
- Spinning with both wheels
- Making an “arc” in either direction.

Possible Approaches

We are using small-scale DC Motors in order to drive our Audio Car, and each motor is uniquely different. They have different rotation thresholds and could possibly spin at slightly different speeds when given the same signals. Because of this, the key issue is not how the car makes these shapes, but how we get the motors to behave in a condition where we can consistently expect them to perform marginally close to each other. If this isn’t the case, then we could face issues with simple actions like drawing a straight line, and inadequate foundations could lead to a negative domino effect when performing the larger “high-level” task.

The simplest approach to fixing motor imbalance is to hard code the duty cycles for each motor in a specified ratio that allows their RPMs to match when at constant duty cycles. This has the possibility of needing recalibration when switching surfaces and could also consume a large amount of time performing trial and error. The car will also need to be at the calibrated speed at all times.

A slightly more complicated approach would be to sample each motor’s RPMs at varying speeds and create a linear transfer function. This approach would allow for the motors to match at different levels of speed but would still need possible recalibration when switching surfaces.

To increase the level of complexity further, we could consistently monitor the RPM of each motor in real time and create a PID system that error-corrects each motor in to meet some sort of RPM goal. This seems like it would solve every problem that the other approaches possess, and while that may be true, the real drawback of this approach is the time it takes to create and debug the PID. It also would perform under the assumption that the optical encoders provide reliable, usable data to the raspberry pi.

Chosen Approach

Unlike software structure, where we try to find a balance between "simple" and "complicated", we treat software functionality in the same manner that we treat hardware functionality, and that is with the mentality of trying the simplest concept first and then building up from it if necessary. With a minimal amount of trial and error, we were able to get both motors to spin at the same RPM using the simplest approach. Because the Audio Cars' tasks don't rely on variable speed, we are able to pick an unchanging duty cycle for one motor and simply find the duty cycle for the other motor that's closest to ideal behavior. This also makes the code incredibly easy to read.

Key Functions

As we explain each key motor control function that the audio car uses, we will start with listing low-level functions. Whenever we have all the low-level functions required to create a high-level task that the user can call, that higher-level function will be displayed. Each function will have a brief description of its purpose above its photo.

Most commands with a "Left" version also have a "Right" version inside driveControls.py, but only the "Left" versions will be shown for simplicity's sake.

The Audio Car will need to travel in a straight line for every Audio Car Command.

```
70  #Makes car move straight for x seconds. Hard-Coded transfer function is used.
71  def straight(seconds):
72      Lmotor.forward(0.725)
73      Rmotor.forward(0.65)
74      time.sleep(seconds)
75      stop()
```

(**Figure 4.5:** "Drive Straight For x Seconds" Function Inside of driveControls.py)

All Audio Car commands require the car to stop after moving.

```
116   #Stops all motors
117   def stop():
118       Lmotor.stop()
119       Rmotor.stop()
120
```

(**Figure 4.6:** “Stop” Function Inside of driveControls.py)

The Audio Car command “Straight” consists of the car driving in a straight line for 3ft. Using the constant speed of both motors set in the previous function, we have calculated that it takes approximately 2.65 seconds for the car to drive approximately 3ft.

```
142   #Full straight command for checkpoint C.
143   def driveStraight():
144       straight(2.65)
145       stop()
```

(**Figure 4.7:** “Straight” Audio Car Command Inside of driveControls.py)

The Audio Car commands “Straight” and “Multiple” call for the audio car to drive straight until it hits the audio source whenever the car is in a condition where it is facing the source and believes that the source has been found.

```
121   #Command for checkpoint B. Moves straight for up to 8 seconds
122   def kamikaze():
123       limSwitch = Button(limSwitchPin, pull_up = False, bounce_time = 0.1)
124
125       Lmotor.forward(0.60)
126       time.sleep(0.04)
127       Lmotor.forward(0.725)
128       Rmotor.forward(0.65)
129       startTime = time.time()
130       while True:
131           if limSwitch.value == 1:
132               stop()
133               break
134
135       print("GOTCHA!")
136
137       limSwitch.close()
138       return time.time() - startTime
```

(**Figure 4.8:** “Drive Straight Until Front Bumper Is Hit” Function Inside of driveControls.py)

The Audio Car commands “Square” and “Figure Eight” require the Audio Car to turn left for a certain amount of time. The “Figure Eight” command also requires the Audio Car to turn right for a certain amount of time.

```
47  #Turning commands for Checkpoint C. Includes both degree and time options.
48  def turnLeft(seconds):
49      Rmotor.forward(0.50)
50      time.sleep(seconds)
51      stop()
```

(**Figure 4.9:** “Turn Left For x Seconds” Function Inside of driveControls.py)

The Audio Car commands “Square” and “Figure Eight” require the Audio Car to move straight after turning. Because of the behavior of our rear “shopping cart” wheel, we have to compensate to make the swivel face straight before the Audio Car moves in the straight direction. If we fail to do so, then the car may diverge from its intended forward direction. So, when making a square for example, we overturn left on purpose, and then we turn the car right for a small amount of time in order to re-center the car and orient the rear swivel wheel to be approximately straight.

```
99  #Straight commands that adjust the robots' rear wheel to go straight after turning
100 def straightAfterTurnL(seconds):
101     Lmotor.forward(0.60)
102     time.sleep(0.10)
103     stop()
104     time.sleep(0.5)
105     straight(seconds - 0.05)
106     stop()
```

(**Figure 4.10:** “Straight After Turning Left For x Seconds” Function Inside of driveControls.py)

The Audio Car command “Square” consists of the car driving in a 3x3ft square and turning left at each corner.

```
147 #Full square command for checkpoint C.
148 def square():
149     straight(2.65)
150     time.sleep(0.75)
151     #0 to 7
152     for i in range(0, 7):
153         turnLeft(1.1)
154         time.sleep(0.75)
155         straightAfterTurnL(2.65)
156         #straight(2.75)
157         time.sleep(0.75)
158     turnLeft(0.85)
159     stop()
```

(**Figure 4.11:** “Square” Audio Car Command Inside of driveControls.py)

The “Figure Eight” command requires the Audio Car to move in an arc at each end, and the car needs to turn a certain number of degrees both before and afterwards to traverse the crossing diagonal lines in the center. By specifying the degrees our car turns, we were able to quickly conceive the behavior we were looking for.

```
53  def turnLeftDegrees(degrees):
54      Rmotor.forward(0.50)
55      #Uses a constant ratio that converts the desired degree turn to seconds
56      time.sleep(0.009555555555555555555555555556*degrees)
57      stop()
```

(**Figure 4.12:** “Turn Left For x Degrees” Function Inside of driveControls.py)

The “Figure Eight” command requires the Audio Car to move in an arc at each end. This relies on each motor being purposefully uneven in rotation speed in order to get the “arc” shape.

```
86  #Arc commands for figure 8 movement in checkpoint C
87  def arcLeft(seconds):
88      Lmotor.forward(0.4)
89      Rmotor.forward(0.60)
90      time.sleep(seconds)
91      stop()
```

(**Figure 4.13:** “Arc Left For x Seconds” Function Inside of driveControls.py)

Both “Multiple” and “Single” Audio Car commands rely on the car spinning in place in order to read the audio environment around it in a manner that maintains equidistance.

```
23  #Spin commands for checkpoint B. Spin motors with no time limit.
24  def spinLeft():
25      Lmotor.backward(0.51)
26      Rmotor.forward(0.51)
```

(**Figure 4.14:** “Spin Counterclockwise Indefinitely” Function Inside of driveControls.py)

The “Figure Eight” Audio Car command relies on the car making a figure eight motion twice. In order to reduce redundancy, this function represents the single traversal of that figure eight function. It uses previously shown functions in order to accomplish the driving path.

```
161  #Does a single figure 8 for checkpoint C. Figure starts in center.
162  def figureEightSingle():
163      time.sleep(0.5)
164      turnLeftDegrees(105)
165      time.sleep(0.5)
166      arcLeft(1.5)
167      time.sleep(0.5)
168      turnLeftDegrees(89)
169
170      time.sleep(0.5)
171      straightAfterTurnL(2)
172      time.sleep(0.5)
173
174
175      turnRightDegrees(100)
176      time.sleep(0.5)
177      arcRight(2.19)
178      time.sleep(0.5)
179      turnRightDegrees(82)
180      time.sleep(0.5)
181      straightAfterTurnR(1.8)
182      time.sleep(0.5)
```

(**Figure 4.15:** “Make One Figure Eight” Function Inside of driveControls.py)

The Audio Car command “Figure Eight” consists of the car moving in a figure eight pattern twice. It starts in the center of the figure eight and must position itself to be at the start of the first “arc” before calling the individual “Figure Eight” function twice.

```
184  #Total figure 8 motion. Includes 2 single figure 8 statements.
185  def figureEight():
186      time.sleep(2)
187      straight(1)
188      figureEightSingle()
189      figureEightSingle()
190      stop()
```

(**Figure 4.16:** “Figure Eight” Audio Car Command Inside of driveControls.py)

Purpose

The “Single” and “Multiple’ commands of the Audio Car require our ADC signals to be interpreted in a way that would allow us to monitor the strength of a targeted frequency and tell the car that it has found a specified audio source. The purpose of signalRead.py is to handle the monitoring of our ADC signals, deliver and modify it in a reliable form of data, and use such data to stop the car in front of the source.

Key Functions

As we explain each key audio analysis function that the audio car uses, we will start with listing low-level functions. Whenever we have all the low-level functions required to create a high-level task that the user can call, that higher-level function will be displayed. Each function will have a brief description of its purpose above its photo.

Taking GPIO Button objects as an input, getStrength serves to sum the values of each ADC digit. Because we didn’t convert the ADC output to binary before sending it to the Pi, we can simply sum them here (similar to counting fingers). This is used to obtain the strength of a given ADC at any instance. This is primarily used in the clock function.

```
#getStrength takes three Button objects and maps them to an array.
#The returned value is the 4th value in the array, which is the sum of the first three binary values.
def getStrength(sig1, sig2, sig3):
    #sigArray is the main data input array. [s1 status, s2 status, s3 status, total strength]
    sigArray = [0, 0, 0, 0]
    if sig1.value == 1:
        sigArray[0] = 1
    else:
        sigArray[0] = 0
    if sig2.value == 1:
        sigArray[1] = 1
    else:
        sigArray[1] = 0
    if sig3.value == 1:
        sigArray[2] = 1
    else:
        sigArray[2] = 0

    sigArray[3] += (sigArray[0] + sigArray[1] + sigArray[2])
    return sigArray[3]
```

(**Figure 4.17:** “Get ADC Strength” Function in signalRead.py)

The clock function serves to digitally smooth out the ADC data in order to create a better sense of the audio source’s strength. It’s also used to mitigate any fast spikes and hardware anomalies. It takes Button objects and “cCount”, an integer, as input. “cCount” refers to the number of clocks being read before returning the average value. A higher clock count will lead to smoother

data, but a longer time between those readings. This function is heavily relied upon in both “Single” and “Multiple” Audio Car commands.

```
31  #"clock" returns the average of the strength values over a given number of readings (cCount)
32  #s1, s2, and s3 are Button objects connected to the ADC GPIO ports.
33  def clock(s1, s2, s3, cCount):
34      count = 0
35      strengthSum = 0
36
37      while count < cCount:
38          strengthSum += getStrength(s1, s2, s3)
39          count += 1
40
41      avg = strengthSum/count
42      return avg
```

(**Figure 4.18:** “Clock ADC Strength” Function in signalRead.py)

The calibrate function is called from the “Debug” input on the dipswitch. It runs ADC clocks and prints out the strength of each “clock” for a specified amount of time. This helps us obtain a sense of the Audio Car’s signal response in any environment and calibrate each ADC potentiometer if necessary.

```
61      #Initializes variables
62      calibrationTime = 120
63      startTime = time.time()
64
65      #While the time is under 4 minutes, clock cycles are ran (refer to clock())and printed instantly
66      while (time.time() - startTime) < calibrationTime:
67
68          currentVal = clock(s1 = s1, s2 = s2, s3 = s3, cCount= count)
69          print("Strength for clock: " + str(currentVal))
70
71      #Closes Button Objects
72      s1.close()
73      s2.close()
74      s3.close()
```

(**Figure 4.19:** Key Section of “Calibrate” Function in signalRead.py)

The function “checkpointB” serves as the main function for the first part of “Multiple” and the entire searching section of the “Single” Audio Car command. The only input it takes is the approximate amount of time it takes for the car to spin 360 degrees counterclockwise. It selects the C8 ADC as its signal input and creates button objects.

From there the car rotates 360 degrees left while keeping track of the maximum value and the time the car found that maximum value. From there it will stop and then spin left for the amount of time it takes to reach the orientation it found that maximum value.

We also have buffers at the beginning and end of both of these actions where the car will rotate 180 degrees left in order to eliminate the amount of time it takes for the car to start and stop.

The intended end behavior is that the car stops while facing the audio source.

```
    tDiff = 0
    max = 0
    maxTime = 0
    driveControls.spinLeft()
    time.sleep(fullSpinTime/2)
    sTime = time.time()
    while tDiff < fullSpinTime:
        tDiff = time.time() - sTime
        strength = clock(s1 = s1, s2 = s2, s3 = s3, cCount= 3000)
        print("Signal: " + str(strength) + " | Time: " + str(tDiff))
        if strength > max:
            max = strength
            maxTime = tDiff + fullSpinTime/2 - 0.2

            print("New Max! | Signal: " + str(strength) + " | Time: " + str(tDiff))

    time.sleep(fullSpinTime/2)
    driveControls.stop()

    time.sleep(3)

    driveControls.spinLeft()
    time.sleep(maxTime * 0.96)
    driveControls.stop
```

(**Figure 4.20:** Key Section of “Checkpoint B” Function in signalRead.py)

The function “checkpointA” serves as the main function for the second part of the “Multiple” Audio Car command. While we would have preferred to reuse the function for “Single”, we unfortunately ran into issues with our C6 signal that created different behavior from the C8 signal, and therefore, we felt that a different software solution was the right choice to make.

GPIO Button objects are created at the ADC outputs of a specified checkpoint. From there, the car starts rotating right for 6 seconds. During this time, the car is recording its top 3 maximum values. Once the car finishes scanning, it averages those values to create a “goal”. This “goal” is what needs to be met in order for the car to have found its source.

```
98      #For 6 seconds
99      while time.time() - sTime < 6:
100
101         #Get value and check if its a new max. If so, populate into array and shift other items over.
102         currentVal = clock(s1 = s1, s2 = s2, s3 = s3, cCount = 5000)
103         print("FINDING MAX... CURRENT VALUE: " + str(currentVal))
104         if currentVal > maxArray[0]:
105             for i in range(2):
106                 point = maxArrayLen - i - 1
107                 maxArray[point] = maxArray[point - 1]
108
109             maxArray[0] = currentVal
110
111             print("NEW MAX FOUND!!  MAX ARRAY: " + str(maxArray))
112
113     driveControls.stop()
114
115     #Sort through array to get rid of outliers & zeros before finding average
116     if maxArray[0] > 1.5 * maxArray[1] or maxArray[0] > 1.5 * maxArray[2]:
117         if maxArray[1] == 0 and maxArray[2] == 0:
118             max = maxArray[0]
119         elif maxArray[2] == 0:
120             max = (maxArray[1] + maxArray[0]) / 2
121         else:
122             print("Max Binned for being Outlier...")
123             max = (maxArray[1] + maxArray[2]) / 2
124     else:
125         max = (maxArray[0] + maxArray[1] + maxArray[2]) / 3
126
```

(**Figure 4.21:** Part 1/2 of “Checkpoint A” Function in signalRead.py)

The car then rotates left for 6 seconds to untangle its cables. It then stops and begins rotating right. During this time, it is running clocks to obtain the ADC strength and comparing it to the previously obtained “max signal”. After every 2 seconds pass (Approximately a full rotation), the car decreases the threshold to count as a “found signal”, thus making the car more sensitive with every rotation. The end behavior is that the car stops whilst facing its “found source”, albeit with a slightly more complicated approach than the C8 source. If it hasn’t found a source in 8 seconds, the car will automatically stop in order to prevent the cables from getting too tangled.

```
    #Start spinning right. Begin "Finding Audio Source" Stage
    driveControls.spinRight()
    decreaseFlag = 0

    sTime = time.time()

    #For 8 seconds, clock values and check if (current value * scalar) exceeds a given threshold integer
    #If so, check again and see if new value exceeds 80% of the original threshold. If so, stop the car because source is found.
    #The number to beat is multiplied by a value less than 1 every 2 seconds in order to increase sensitivity every rotation.
    while time.time() - sTime < 8:

        if time.time() - sTime > 2 * (decreaseFlag + 1):
            print("INCREASING SENS")
            decreaseFlag += 1
            numberToBeat = numberToBeat * decayRate

        currentVal = clock(s1 = s1, s2 = s2, s3 = s3, cCount = 5000) * scalar
        print("Clocking Value: " + str(currentVal))

        if currentVal > numberToBeat and currentVal < ceil:
            currentVal = clock(s1 = s1, s2 = s2, s3 = s3, cCount = 4000) * scalar
            print(" - Verifying Max: " + str(currentVal))

            if currentVal > numberToBeat * 0.9 and currentVal < ceil:
                time.sleep(0.1)
                driveControls.stop()
                maxFound = 1
                break
```

(**Figure 4.22:** Part 2/2 of “Checkpoint A” Function in signalRead.py)

4B.3 - Main User I/O

Purpose

When the user starts up the car, they use a dipswitch to communicate to the car which command they want to execute. The purpose of readAndRun.py is to take in those dipswitch values and appropriately call the desired command in a manner that is friendly to the user. The car must also

be able to call multiple commands in succession without instantly running a command consecutively.

Programming Walkthrough

Unlike the other two programs, which act as function libraries, readAndRun.py acts more as a script. The only function inside readAndRun.py serves to call commands based on its string input. Because of this, we will display that function below and then walk through the functionality of the rest of this program.

Through a large chain of if statements, commandCaller uses an input string to initiate one of the available Audio Car commands. Because some use LED's or return values from other functions, they may take up more space than simple function calls to other files. That being said, a large majority of this function is calling functions from signalRead.py and driveControls.py, which was the intended outcome of our software structure.

```
#Calls other python functions based on string representation of dipswitch status
def commandCaller(commandString):
    if commandString == "Straight":
        driveControls.driveStraight()
    elif commandString == "Square":
        driveControls.square()
    elif commandString == "Figure 8":
        driveControls.figureEight()
    elif commandString == "Switch Test":
        driveControls.kamikaze()
    elif commandString == "Single":
        signalRead.checkpointB(fullSpinTime=1.79)
        driveControls.stop()
```

(**Figure 4.23:** Example Section of Command Caller Function in readAndRun.py)

After defining commandCaller, we initialize the GPIO objects of the dipswitch inputs and LED outputs. We also initialize a dictionary that links the dipswitch values (in string form) to their corresponding command.

```
#Define dictionary that maps bitmap to its command definition
commandArray = {
    "0000" : "Don't Move",
    "1010" : "Debug",
    "1011" : "Switch Test",
    "1000" : "Single",
    "1001" : "Multiple",
    "1100" : "Straight",
    "1101" : "Figure 8",
    "1110" : "Square",
    "1111" : "Off"
}
```

(**Figure 4.24:** Command Dictionary Initialization in readAndRun.py)

After initializing all necessary variables, the car will flash all rear LEDs twice to indicate that the car is receiving the dipswitch signals and is therefore ready to interpret a command. Then, as long as 10 minutes haven't passed, the car will constantly be reading the Dipswitch values and display those values on the LEDs in the same orientation. It also constantly checks if the command is the same. If a command hasn't changed for 7 seconds, the program will interpret that as a call and will use commandCaller to initiate the desired command.

If the runtime exceeds 10 minutes or the command "1111" is sent to the car, the program will end, closing every relevant GPIO port on its way out.

```
#Program runs indefinitely until 10-minute time limit is reached or the "Off" command is called
while (time.time() - starttime) < 600:
    bitString = ""

    #Obtain bit values and display LED representation of bitmap
    for i in range(0, 4):
        if gpioArray[i].value == 1:
            bitArray[3 - i] = 1
            ledArray[i].on()
        else:
            bitArray[3 - i] = 0
            ledArray[i].off()

        bitString += str(bitArray[i])

    if bitString == "0000":
        calledCommand = ""

    #Checks if the status of dipswitch has changed. If so, resets 7-second command debounce
    if (bitString != oldBitString):
        oldBitString = bitString
        print("Dipswitch Value Changed!: " + bitString)
        try:
            print("Command: " + commandArray[bitString])
        except KeyError as e:
            pass
        sameCommandTime = time.time()

    #If valid command is held for 7 seconds, Strobe LEDs and run commmand.
    if ((sameCommandTime - time.time()) < -7) and (bitString != "0000") and commandArray[bitString] != calledCommand:
```

(**Figure 4.25:** Main Behavior in readAndRun.py)

Overall Conclusion

Considering everything that needs to be accomplished within the Audio Car, we believe that there have been adequate amounts of consideration for reducing space/complexity where

possible whilst still completing all necessary tasks to achieve the Audio Car's behavior. The most complicated section (on both user and developer ends) is definitely the program for checkpoint A, which is ultimately a software solution to hardware issues. While we are glad to have completed checkpoint A-, we would definitely approach our hardware differently in order to mitigate software complexity for this specific checkpoint and possibly the checkpoint B function as well. It would be ideal for both frequencies to have the same algorithm as that would make things easier for everybody.

Chapter 4C – Software: Running on Startup

Purpose

As both a project requirement and a thoughtful consideration from the end-user's perspective, the car must operate upon startup without any sort of outside connection, command, or input. This

means that the main python file (readAndRun.py) must be automatically called after the Raspberry Pi has finished booting.

Research & Chosen Approach

When beginning our research on selecting our startup approach, we came across a multitude of options. The first option was to implement a crontab file to run on startup. Another option was to modify the pre-existing .bashrc file (runs on startup) to run the main python program. We are also able to do a similar modification to rc.local on the Pi.

Because members in our group already had experience with modifying .bashrc from past experiences, we decided to go with what we already knew. The file ".bashrc" runs once on startup and mainly contains information about how the terminal is displayed to the user. By writing a collection of Unix commands at the end of this script, we have not only been able to make the Audio Car run on startup, but we have also been able to add other convenient features for testing and programming.

These features include:

- Displaying the current IP to ssh into (FabLab IP changed every couple days)
- Navigates to the Code directory in the git repository on startup
- Displaying the current Code directory and subdirectories in terminal

Program

```
67 ## OUR MODIFICATIONS TO .bashrc
68 ## Display current IP Address in case IP changes. Display -I multiple times in case of cutoff.
69 echo "IP ADDRESS TO SSH TO: "
70 hostname -I
71 hostname -i
72 hostname -I
73
74 ## Cd to current git branch and code directory
75 source ./car/bin/activate
76 cd /home/group11/Group-11
77 cd Code
78
79 ## Print out current code directory
80 echo "Current Code Directory: "
81 tree
82
83 ## Run main python file for running Audio Car Commands
84 python ./Implementation/src/readAndRun.py
```

(**Figure 4.26:** Modifications to .bashrc on the Raspberry Pi.)

```
Linux group11 6.6.51+rpt-rpi-v8 #1 SMP PREEMPT Debian 1:6.6.51-1+rpt3 (2024-10-08) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Apr 16 15:13:14 2025
IP ADDRESS TO SSH TO:
172.20.232.70
127.0.1.1
172.20.232.70
Current Code Directory:
.
├── Implementation
│   └── src
│       ├── driveControls.py
│       ├── __pycache__
│       │   ├── driveControls.cpython-311.pyc
│       │   └── signalRead.cpython-311.pyc
│       ├── readAndRun.py
│       └── signalRead.py
├── Testing
│   ├── figureTest.py
│   ├── gpioSequence.py
│   ├── MotorReader.py
│   ├── MotorTest.py
│   ├── readSwitchboard.py
│   ├── RPMs.csv
│   ├── singleLEDTest.py
│   └── switchTest.py
```

(Figure 4.27: Terminal Output on Raspberry Pi Startup)

```
Dipswitch Value Changed!: 0011
Dipswitch Value Changed!: 1111
Command: Off
Command Called!
Off
) group11@group11: ~/Group-11/Code [main*] $
```

(Figure 4.28: Terminal Output of readAndRun.py on Raspberry Pi Startup)

Testing

- Confirm that startup is consistently activated
- Confirm that bash and python program outputs are being displayed

Conclusion

Modifying .bashrc led to everything being called consistently while being incredibly convenient to modify when necessary. Unless the git repository was corrupted, everything would print out as displayed. We were able to already be in the relevant Code directory on startup and see the location of all files if they needed to be accessed. This would end up saving us a great deal of time while testing and debugging the audio car.

Chapter 5 – Ethics

Unique Design

Throughout this project, our team prioritized original design, academic honesty, and personal understanding of every system component. While we referred to public resources—such as circuit calculators and example schematics for the push/pull amplifier, filters, and ADC—we heavily modified these designs to meet the unique requirements of our audio tracking car. We made sure that the versions we implemented were tailored for our specific voltage levels, frequency ranges, and integration needs.

Work was divided clearly among team members on a weekly basis, using spreadsheets to assign and track responsibilities. Deliverables were completed collaboratively, with every member contributing either independently or in small subgroups. Even when software or hardware tasks were unevenly distributed, members cross-trained and assisted one another to ensure consistent contribution across the board. Our GitHub repository provided additional tracking for the software development process.

Though we occasionally sought feedback from other teams or instructors—particularly regarding the push/pull amplifier—this help was used strictly for design review, not copying. We ensured we understood every design aspect we implemented and avoided sharing or requesting full circuits or code from others. Any tools or references used were properly cited in the Appendix.

By maintaining clear separation between inspiration and implementation, we ensured our work was authentic and rooted in our own understanding, not simply replicated from existing designs.

Ethical Considerations

If our audio tracking car were developed into a commercial product, it would raise some ethical concerns related to safety, privacy, reliability, and accessibility.

First and foremost, there are potential physical hazards: exposed wires could short if improperly handled, powering the Raspberry Pi before the push/pull amplifier could create voltage instability, and liquid exposure could damage the circuitry. These issues could be mitigated through full PCB integration, robust enclosures, and user-friendly assembly instructions.

From a functional reliability standpoint, the car uses narrow-band filters to distinguish audio tones. If the system misidentified a signal, it could drive toward the wrong source. While the bump switch would stop the car upon contact, such misdirection could cause confusion or reduced trust from users. Robust filtering and more advanced ADC logic would be important in a consumer-ready version.

Importantly, our design raises no privacy concerns. Audio signals are passed directly through filters and never recorded, stored, or sent to the Raspberry Pi. The system performs all detection through analog hardware, ensuring user environments are not monitored in any way.

In terms of maintainability, our prototype relies on a breadboard circuit that would be replaced by a PCB in a commercial version, improving robustness and reducing failure points. In the event of malfunction, repairs would ideally be handled by trained technicians.

The product is generally accessible to a wide audience, including those with hearing impairments. It only requires the ability to play an audio tone near the car. Physically, a user would need to plug in the Raspberry Pi and 24V supply, possibly from the floor level, which could present minor accessibility concerns in its current form.

While environmental impact was not a major consideration in our project, mass production would benefit from attention to recyclable components, minimal e-waste, and energy-efficient operation.

Finally, our team consistently followed principles aligned with the NSPE Code of Ethics, including:

- Prioritizing user safety through circuit protection and bump sensor logic
- Maintaining honesty in all documentation and reporting
- Working within our areas of competence, while seeking guidance only to clarify or improve our understanding

Chapter 6 – Self Critique

Teamwork

Our team maintained consistent and effective communication throughout the semester. With only four members, we kept coordination simple by using a shared iMessage group and Excel spreadsheets to track tasks and progress. Roles were understood and respected, but we stayed flexible, for instance, our programmer assisted with hardware when needed, and vice versa.

Team meetings occurred 2–3 times per week, either as a full group or in subgroups, and we were able to focus intensely while also building strong friendships. There were no major disagreements; the most debated topic was how to format the technical document, and even that was resolved quickly. Michael See, in particular, took initiative during critical testing periods—especially in the final days before checkpoints—often working long hours to refine the system with support from the rest of the team.

Hardware

Most of our hardware designs required several rounds of revision. Very few components worked perfectly on the first try, and every major subsystem was iterated on. The push/pull amplifier proved the most difficult; while it initially worked in isolation, it failed to maintain +/-12V once integrated with the rest of the car. This required extensive redesign and troubleshooting, made more challenging by limited reference material and few comparable examples.

Our final hardware layout was functional but physically compact to the point of being cluttered. Unlike many teams who used multiple layers of breadboards or protoboards, we kept everything on a single layer. While this gave us the smallest and most compact car overall, it came with downsides—tight wire spacing occasionally caused component contact issues, particularly around the ADC and filters.

Looking back, we would have benefited from a higher-resolution ADC and a transistor switching mechanism between filter channels. Additionally, moving some components to protoboard or a PCB would have increased circuit stability and reduced debugging time.

Software

Our software was well-structured and thoroughly commented. Variables and functions were appropriately named, and our GitHub repository made day-to-day collaboration smooth. We received strong feedback from instructors during code checks, particularly on clarity and documentation.

One persistent issue was Git corruption during testing sessions, especially when the Pi attempted a git pull on startup. This led to us disabling that feature for stability. Initial software performance was rough, particularly around frequency detection logic, but consistent testing and incremental revision allowed us to refine our algorithms.

By the end, DIP switch mode selection and startup logic worked reliably, and the car responded consistently to each command. While the software could always be improved, most limitations we encountered stemmed from hardware constraints, not algorithm design.

Other Critiques

The compactness and aesthetic cleanliness of our car was one of our proudest achievements. While other teams had tangled wiring and bulky chassis designs, we were able to maintain a clean single-layer layout without sacrificing function. We believe we had the smallest fully functional car by the end of the semester.

One challenge we didn't anticipate was the amount of work outside the physical project—presentations, weekly paper deliverables, and mid-semester exams all competed for time and energy. Despite that, we managed our time well overall, consistently submitting deliverables early. The only exception was Checkpoint A-, which we passed two days after the official deadline due to late-stage debugging. Still, we dedicated significant effort in those final days to meet expectations.

Our documentation process started strong but became somewhat rushed near the end. After Checkpoint A-, we made the decision to completely reformat the technical document. This transition has been a bit chaotic, but thanks to our earlier documentation, we haven't lost any critical information.

If we could offer future teams one piece of advice: prioritize hardware early. Clean hardware will make the software easier to develop, while trying to compensate for hardware issues with code is far more time-consuming and frustrating.

Chapter 7 – Revision Plan

Results

Overall, the car did fairly well, but there is definitely room for improvement. The final checkpoint passed late because the car could not detect the two different frequencies in one run, and it had trouble starting up sometimes. When the car did detect the frequency, sometimes it would be facing the opposite direction or slightly off from the sound. Here are some possible changes that could be made.

Possible Changes

PCB: If we were to implement our PCB design on the car it would become much cleaner and less susceptible to noise interfering with the circuit. The breadboard is more susceptible to noise from loose connections and longer traces from the wires so this could help the accuracy of the car.

Microphones: The microphones we used are very cheap and inconsistent. They are prone to distortion and clipping. This gets even worse when we use multiple as the two microphones we used didn't have the same responses. If we had more expensive microphones, they would be more accurate and have a flatter frequency response. We could also place something on the back of the car to help block noise reaching the back and the microphone detecting the sounds in the opposite direction.

Real Product Changes

If this were a real product, we would be using our designed PCB and higher quality equipment. The code also used two different algorithms when finding the two different frequencies which could be confusing for customers or consumers who bought the car. The car could also be worked on to start up faster so the consumer would not have to wait as long to use the car. We would also have to install batteries into the car so no one would have to stay near an outlet to use it.

Chapter 8 – Individual Reflections

Dawson Gulasa

“This project has been one of the most rewarding experiences of my academic journey so far. At the start of the semester, the idea of building a fully autonomous audio-tracking car felt almost impossible, but watching the hardware and programming come together as a final, functioning product was incredibly satisfying. I especially enjoyed the collaborative aspect, our team supported each other from day one, and it genuinely felt like we were working toward a common goal rather than just completing an assignment. While I focused primarily on documentation and testing, I also found myself drawn to the interplay between code and circuitry, and I wish I had been more involved in the programming side. That said, working so closely with the Raspberry Pi taught me a ton about real-world embedded systems, and I now feel equipped to pursue many of the project ideas I’ve had in the past. This course changed how I see engineering, not just as something you study, but as something you build. It felt more like working at a company than taking a class, and that has boosted my confidence as a future engineer. I now feel much more prepared for internships, upper-level design courses, and real-world challenges, and I’m excited to take on more projects like this in the future.”

Finn Morton

“I was originally nervous about this class and scared that I wouldn’t be good enough to pass. But this has become the most rewarding class of my career so far. This is the biggest project I have worked on, and I have learned what it's like to fall and rise again through each checkpoint. Being able to research and design our own circuits and use them for real applications has been the most fun I have had at this university. I also got to work with some wonderful people who I now consider friends and look forward to learning with them in future classes. When talking about myself, I do think I could have improved by sharing more of the workload on the documentation and code. I mostly worked on the hardware and put less time into other parts of the project that could have used more help. I have learned not to stick to one design for too long. I spent lots of time trying to fix designs like the push-pull when I should have just redesigned it. Not everything is going to work first try in engineering so I shouldn’t assume my first design will work with enough effort. Sometimes it is easier to start over. Overall, I feel more confident in my abilities as an electrical engineer and am prepared to keep challenging myself in the future.**”**

Cade Toney

“This class has been my favorite class I have taken in my academic career. I’ve never worked on a project this big in my life or experienced this much failure. This class and project pushed me harder than I’ve ever been pushed, it was so rewarding. Getting to research, design, test, and document a real engineering project was fun. The project was really cool, especially given the fact that I am passionate about filtering and is actually what got me into electrical engineering. I really like my team and gained some real friends through working together. There were some things I feel like a came up short on though. I wish I had done better helping with documentation as there were some weeks I didn’t help with the write up at all. I still feel like work was evenly distributed, but it was sometimes unrewarding to just work on the project alone and not help with other things. And even though I worked on a lot of different designs on the car, a lot of them didn’t work first try, or work sufficiently. This led to a distaste in my mouth because even though I had designed a lot of things they ended up not making it on the car. However; this is the reality of engineering and like the cat in the hat said, “sometimes to find out where something is, you have to find out where it's not” and I feel like I lived by this during this semester. Overall, the project was a joy and so was the class. I love my team and our final product!”

Michael See

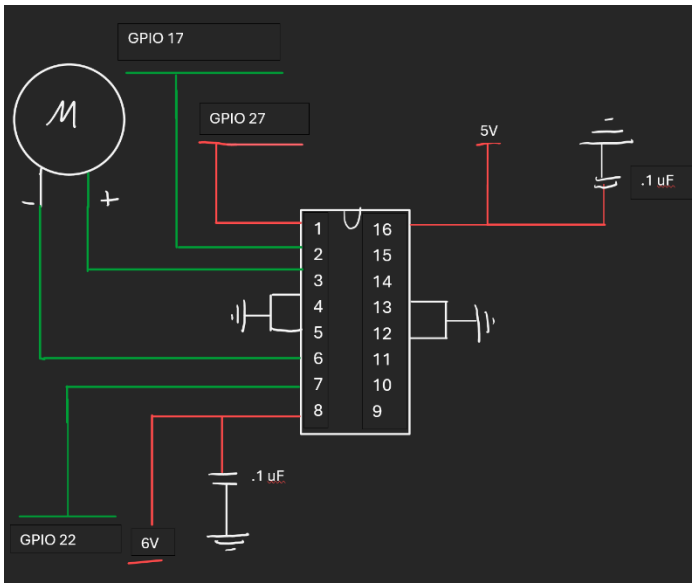
“When signing up for this class, my advisor warned me of its challenges. She specifically told me that this is the class that has people in Driftmier until 4AM. She had also told me that Circuits was difficult, but I found it to be straightforward, so I brushed off her comment about design. Having now finished this class, and left Driftmier at a collection of ungodly hours, I can safely say that I should’ve taken her advice a LOT more seriously, and my previous dismissal was out of ignorance of what was to come. I’ve done similar projects in the past that have taken time, blood, sweat, and tears to complete but I have never done so in a group. This was the biggest takeaway that I got from this class, that engineering is truly a team-based career. What comes with teamwork is communication, accountability, confidence, and trust. I started out this class just doing my part and that’s all. I had mild concerns for any issues or obstacles that others in my team came across because it ‘wasn’t my job’, but that’s not how we pass. Sometimes things work on the first try, and sometimes the whole project will be bottlenecked by a single component or caveat. When everyone is assigned a task and it comes across as ‘you do x and you do y’, the weight of each task is sometimes shrouded by the fact that in that moment it is a goal, a word, a concept. As time moved on, I realized that we worked much better as a team if we tackled things dynamically and allocated ourselves as needed. This meant that I had to get out of my comfort zone and try to help others with things I told myself I wasn’t good at, which ended up not only expanding my knowledge in those fields, but also leading to everyone in my group helping me with programming and testing, which was the only reason we can say today that we passed checkpoint A-.

I’ve found it hard to completely trust others with tasks that affect me, and my coping mechanism is to just take over and do it myself. This is an unsatisfactory practice in engineering for a multitude of reasons, and while this semester wasn’t completely void of that happening, I learned

to understand that I'm in a time and place in my life where those around me are just as (if not more) intelligent and capable than I am. If I did this project all over again, I think that I would take a completely different approach now in comparison to how I started this winter. Personally, I take this as a sign of personal growth and intellectual development, which is all I can ask from any class I take. Thank you, to the professors that pushed us to do better, the TA's that saved our rear ends, and my teammates who suffered alongside me"

Chapter 9 - Appendix

A.A1- Motors and H-Bridge: Previous Designs



(**Figure 9.1:** First iteration of the H-Bridge circuit, with the enable controlled by GPIO.)

A.A2- Motors and H-Bridge: Hardware We Did Not Design

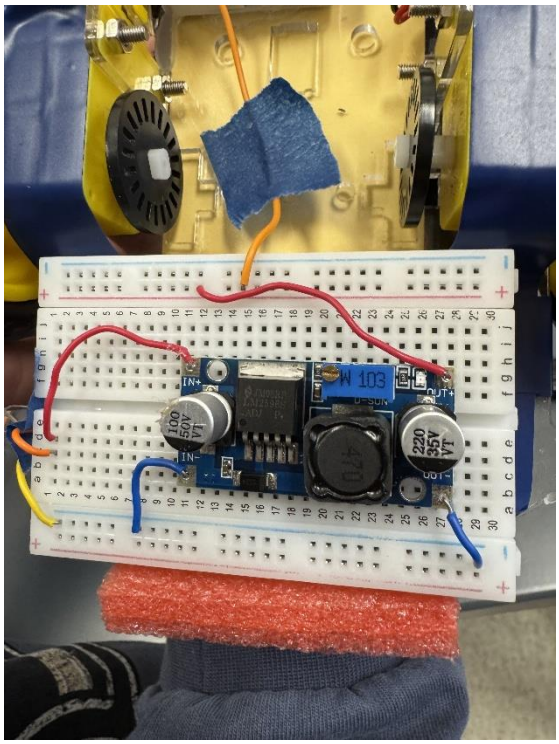
The buck converter was not designed by the group but was provided to the group by the instructors. However, it is worth discussing it.

The buck converter, although a very small part of our audio car design, is very crucial to the operation of our car. It is responsible for providing the 6V input into the H-Bridge required by the motors.

The buck converter consists of the buck converter IC, a potentiometer, and other passive components to step down the input voltage to a specified output voltage whilst keeping the power output constant.

To tune the output voltage to the 6V required for the motors the group used a voltmeter at the output of the buck converter and input the 24V-5A power supply, using a flathead screwdriver to tune the resistance of the potentiometer. The group did this until the output read 6V.

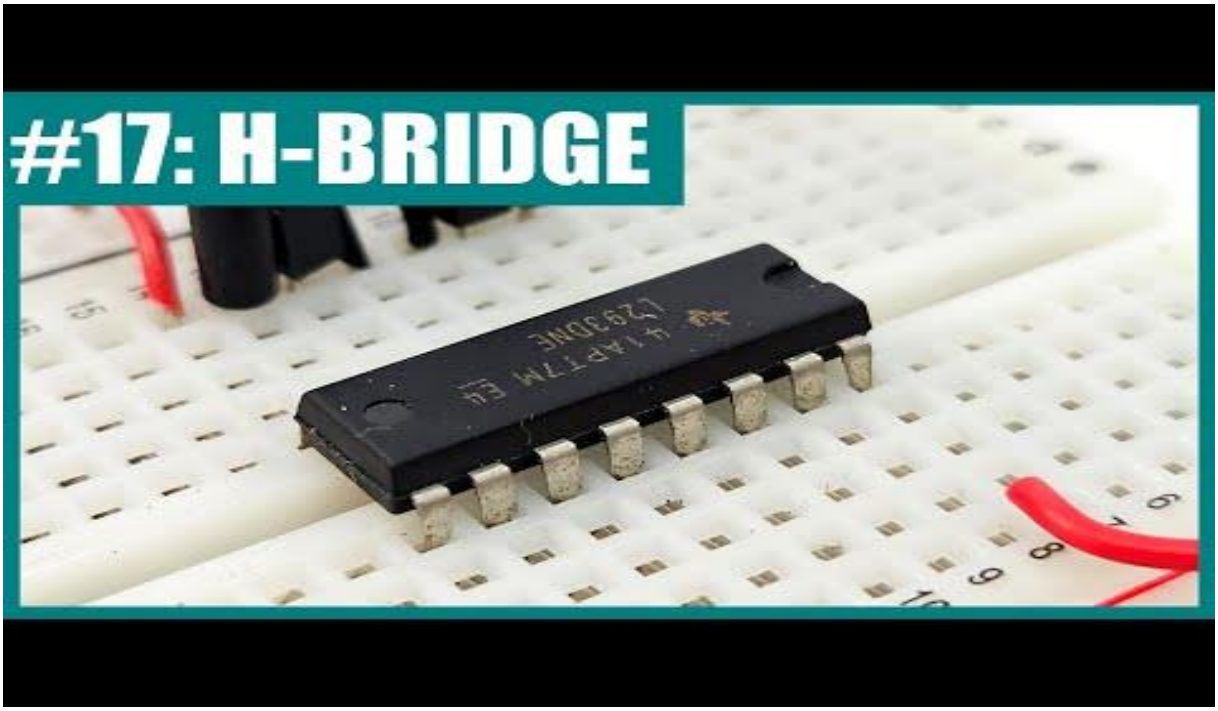
Regardless of the input voltage of the buck converter – as long as it is higher than 6V – the output will always be 6V. This was convenient because the group did not have to alter the buck converter, from CPC using the 24V directly to CHB using the 12V from the positive rail of the push pull amplifier.



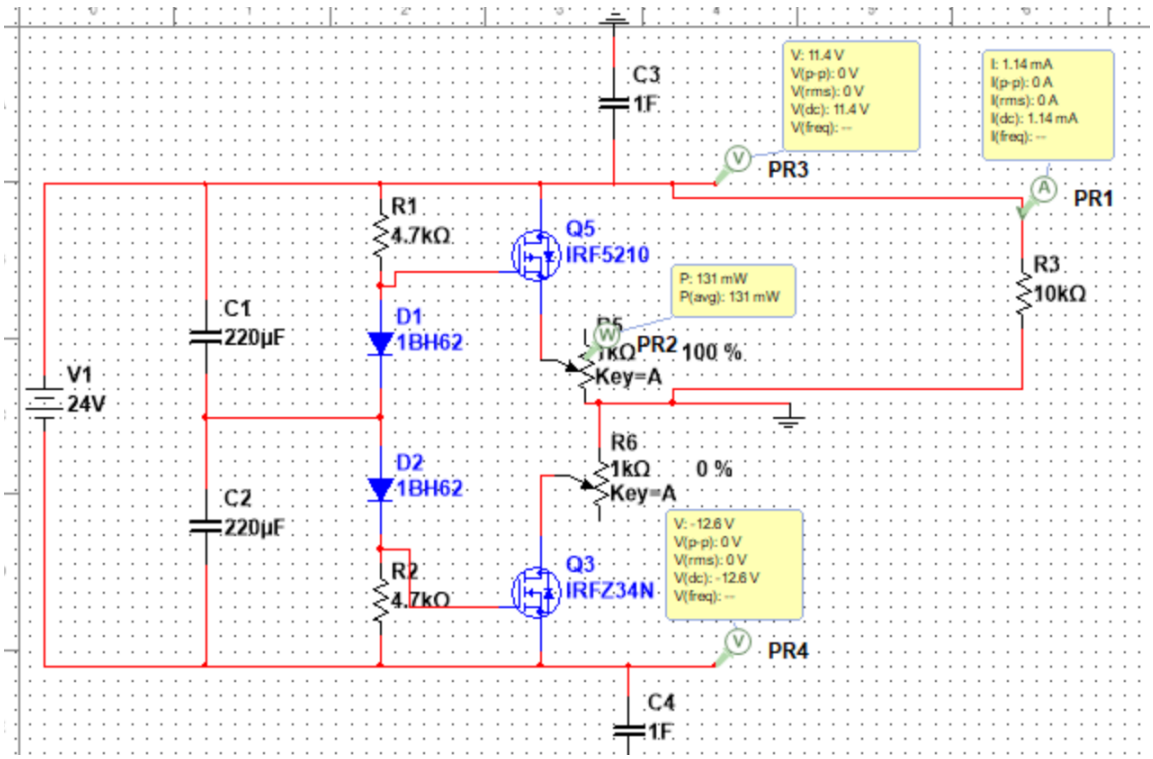
(**Figure 9.2:** Buck converter On Audio Car.)

Link to the YouTube video that help with H-Bridge circuit design:

https://www.youtube.com/watch?v=YU17L650k3s&t=632s



A.B1 - Push/Pull Amplifier: Previous Designs



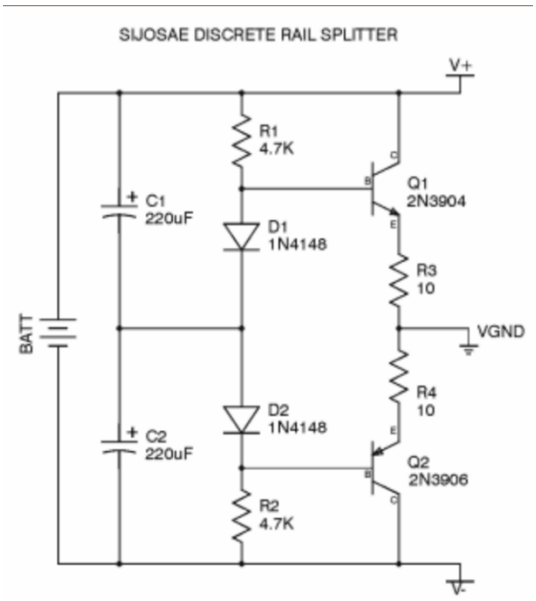
(Figure 9.3: Original Push/Pull design.)

Failed due to not provided a stable enough virtual ground and not supply high enough current for the motors. This design ultimately failed because the Nmos and Pmos transistors are in the wrong location in the circuit and should be swapped. Also, the drains of both transistors to be wired to their respective voltage rail, and the sources of both transistors should be wired to the virtual ground.

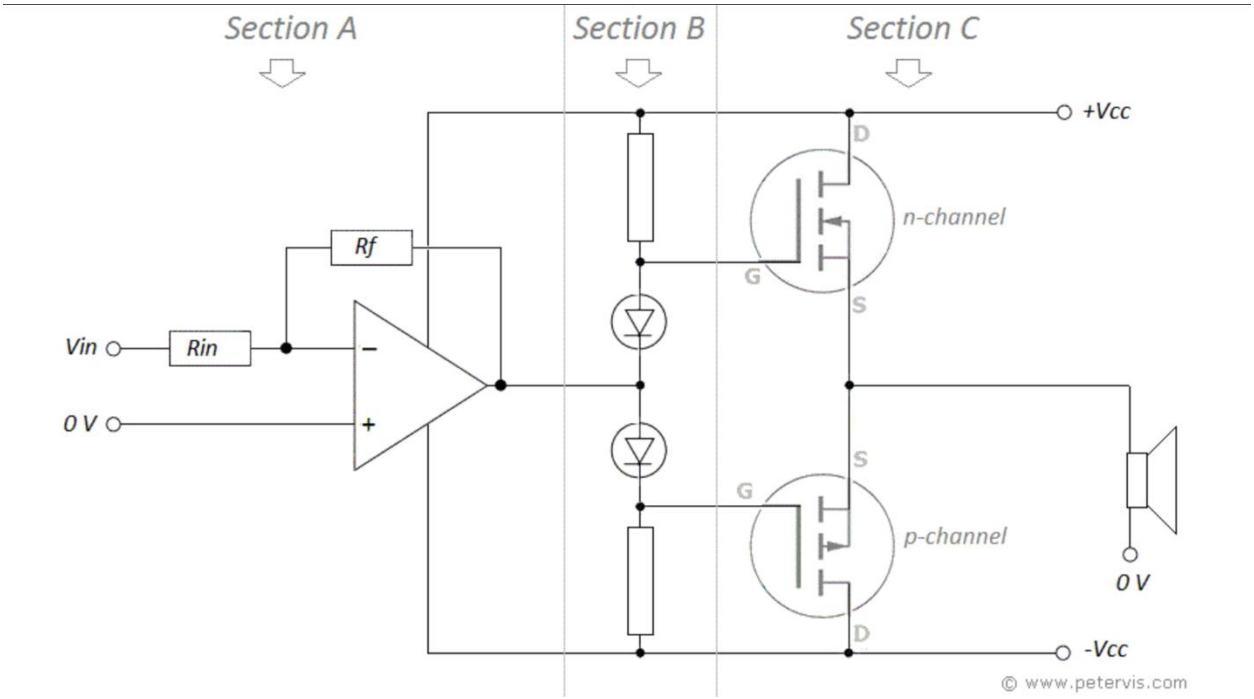
A.B2 - Push/Pull Amplifier: hardware We Did Not Design

The article “Virtual Ground Circuits” which was used to research the initial push pull amplifier design:

https://tangentsoft.com/elec/vgrounds.html



(**Figure 9.4:** Sijosai Discrete Rail Splitter than inspired the group's first Push Pull Amplifier Design.)

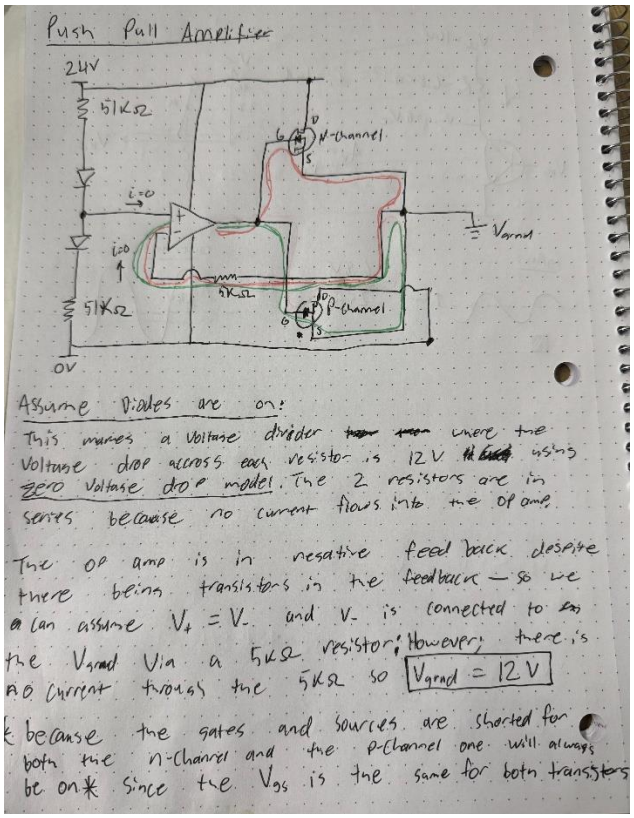


(**Figure 9.5:** Circuit schematic that was the inspiration for the groups final push pull design.)

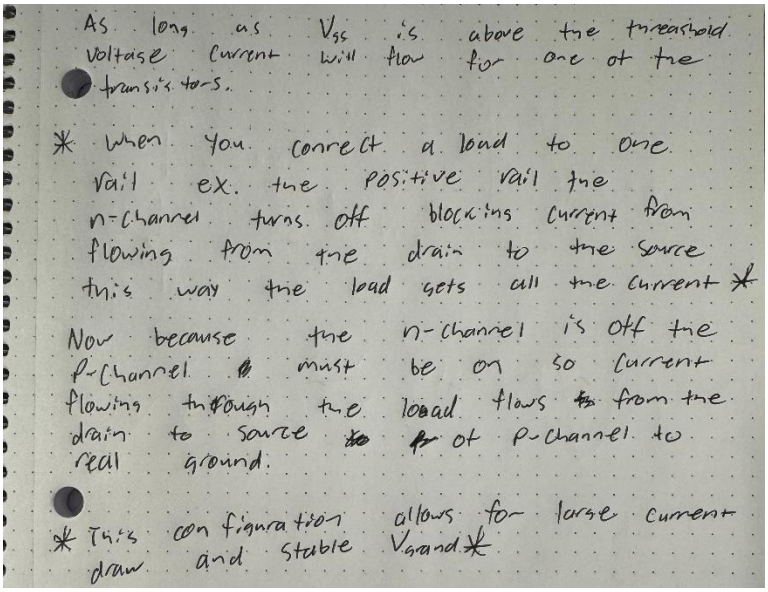
Link to the reddit post that discussed the groups final push pull design:

https://www.reddit.com/r/ElectricalEngineering/comments/1azecfa/how_would_you_make_a_dual_rail_power_supply/

A.B3 - Push/Pull Amplifier: Theoretical Analysis



(**Figure 9.6:** Hand Calculations for Push/Pull Amplifier.)



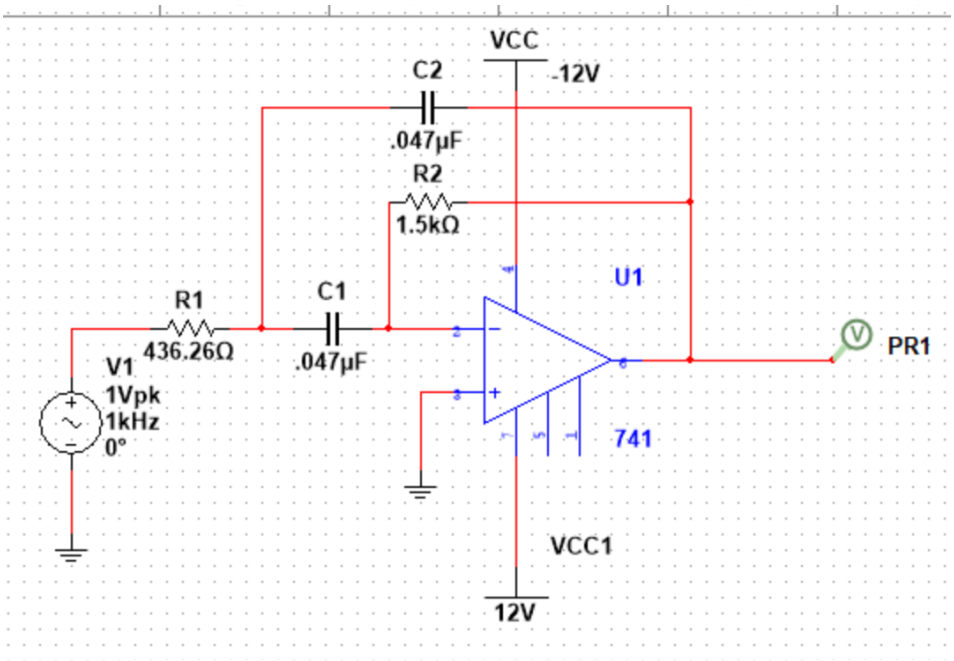
(**Figure 9.7:** Handwritten Theoretical Analysis for Push/Pull Amplifier.)

A.C1 - Filter: Previous Designs

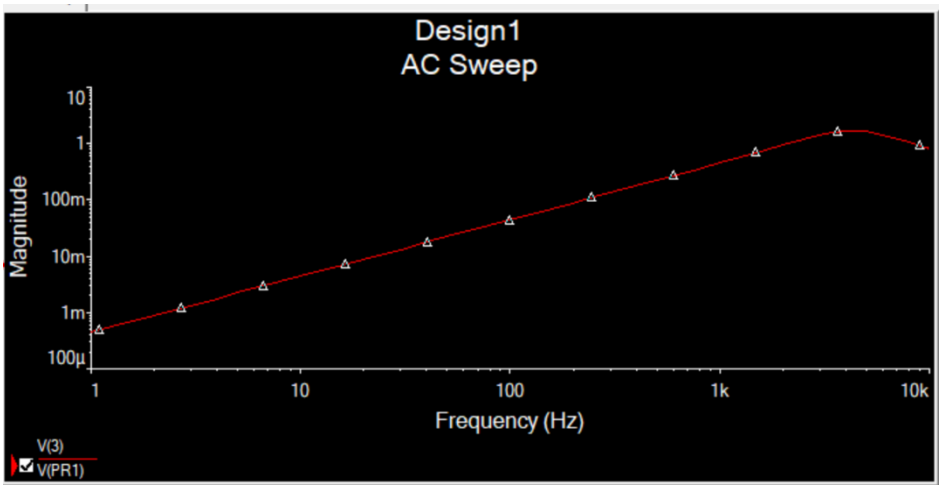
Infinite Gain Multiple Feedback Active Filter

Purpose: This was the first attempted design due to its high theoretical Q-factor and gain.

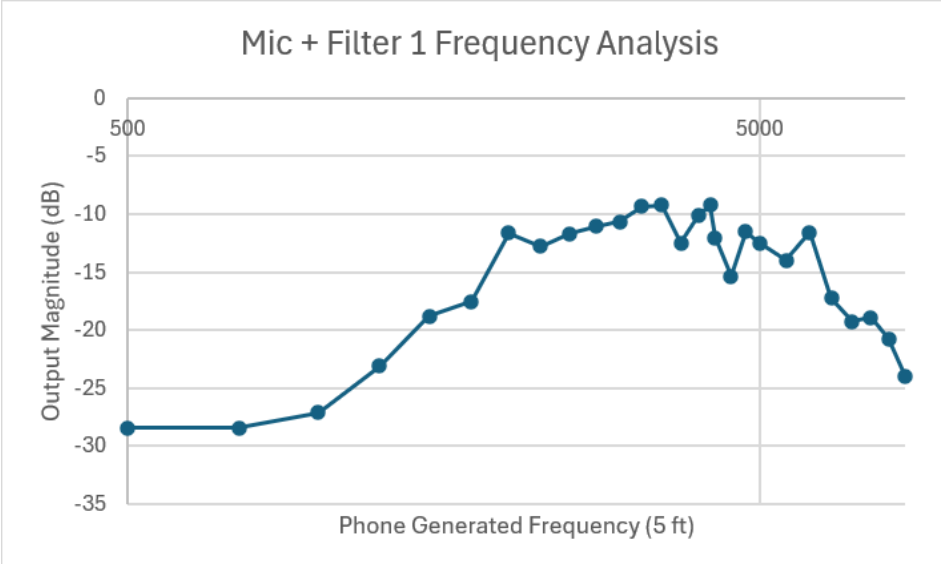
Outcome: Failed to produce consistent results with real microphone input. Multisim performance did not translate to physical stability, so the design was abandoned in favor of the narrower, more stable dual-stage bandpass filter.



(**Figure 9.8:** Constructed Infinite Gain Multiple Feedback Active Filter in Multisim.)



(**Figure 9.9:** Simulated Bode plot of Infinite Gain Multiple Feedback Active Filter in Multisim.)



(**Figure 9.10:** Physical Bode plot of Infinite Gain Multiple Feedback Active Filter with Microphone Input.)

A.C2 - Filter: Hardware We Did Not Design

Online Filter Design Calculator

The bandpass filter topology and component values were obtained using the following tool: https://www.calctown.com/calculators/narrow-bandpass-design

The calculator outputs ideal R and C values based on desired center frequency, gain, and Q factor. While we adjusted some values for practical implementation and tuning, the underlying structure was not our own original invention.

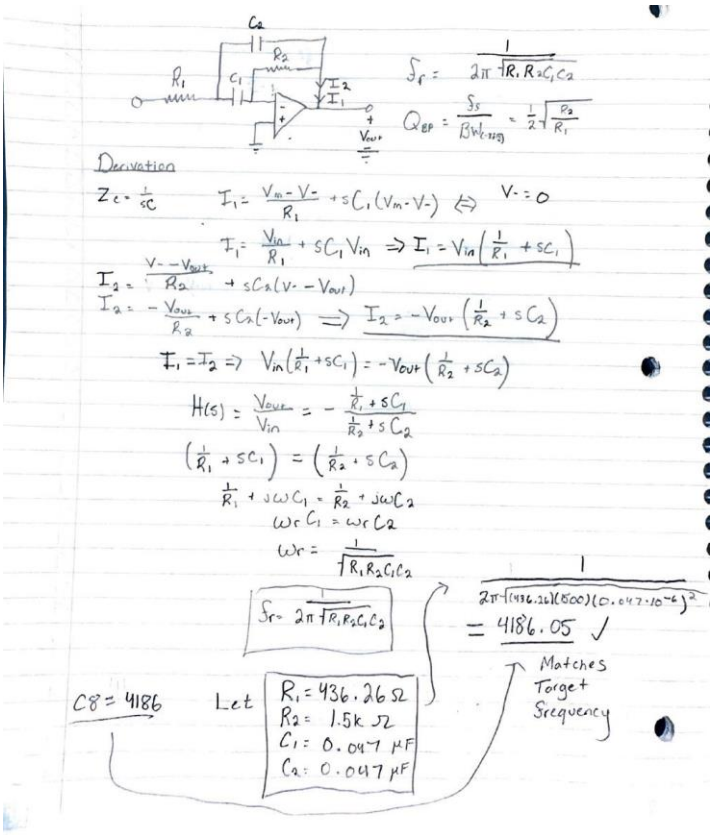
A.C3 - Filter: Theoretical Analysis

The following equation was used to determine the center frequency of our filters:

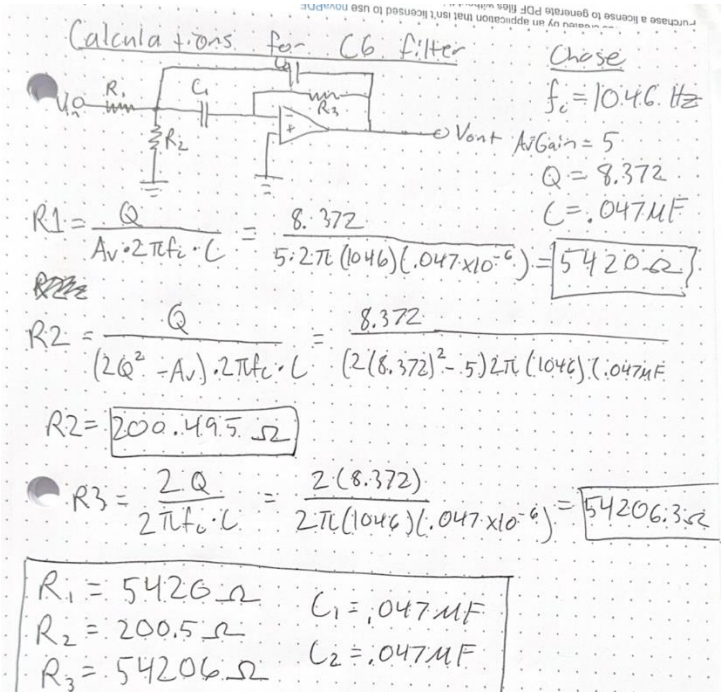
$$fr = \frac{1}{2\pi\sqrt{R1R2C1C2}}$$

We used this equation in both early hand calculations (for both C8 and C6) and online design tools to verify and optimize values.

Below are our hand-calculations for both filters. These values have been scaled and/or altered but this shows our theoretical analysis of both filters.

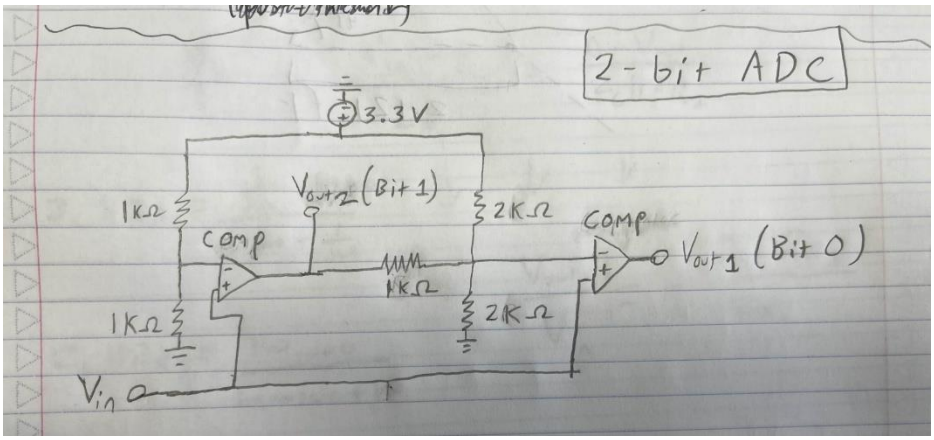


(**Figure 9.11:** Hand Calculation showing Theoretical Analysis for C8 Filter.)

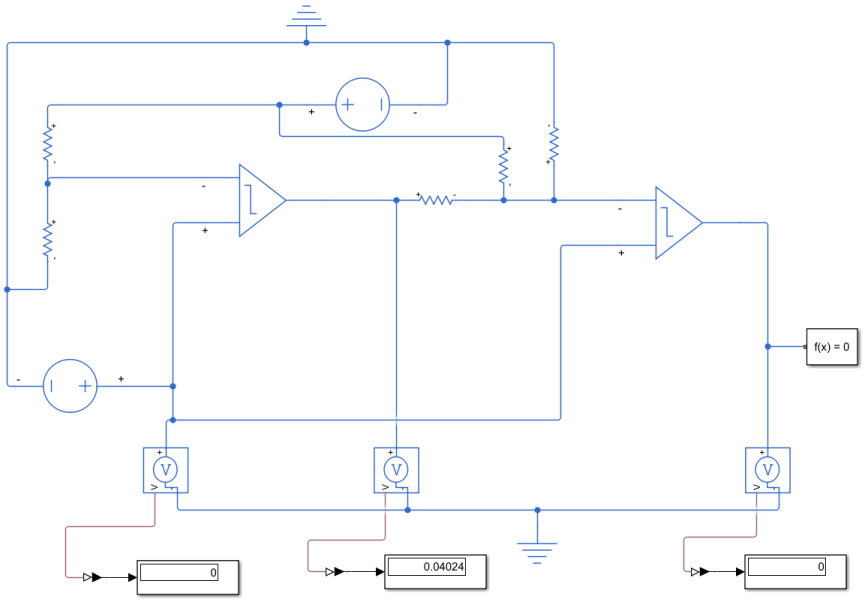


(**Figure 9.12:** Hand Calculation showing Theoretical Analysis for C6 Filter.)

A.D1- ADC: Previous Designs



(**Figure 9.13:** Paper Design for 2-Bit ADC using R-2R ladders and comparators)



(**Figure 9.14:** ADC Simulation Circuit and Result for bitmap 00)